

## Cours d'introduction à la programmation (en Java)

### Types avancés I (en Java)

Jamila Sam  
Jean-Cédric Chappelier  
Vincent Lepetit

Faculté I&C

## Rencontre du 5<sup>e</sup> type

À ce stade du cours, la représentation des **données** se réduit aux types élémentaires `int`, `double` et `boolean`.

Ils permettent de représenter, dans des *variables*, des concepts simples du monde modélisé dans le programme :  
dimensions, sommes, tailles, expressions logiques, ...

Cependant, de nombreuses données **plus sophistiquées** ne se réduisent pas à un objet informatique élémentaire.

- un langage de programmation évolué doit donc fournir le moyen de **composer les types élémentaires** pour construire des types plus complexes, les *types composés*.

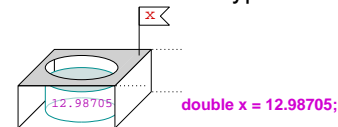
## Exemples de données structurées

Âge	Nom	Taille	Âge	Sexe
20	Dupond	1.75	41	M
35	Dupont	1.75	42	M
26	Durand	1.85	26	F
38	Dugenou	1.70	38	M
22	Pahut	1.63	22	F

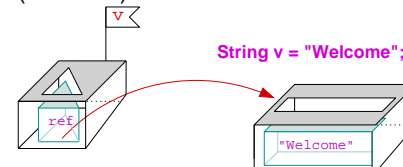
- ▶ tableaux
- ▶ structures de données hétérogènes  
(par exemple, « un enregistrement » dans le tableau de droite ci-dessus)
- ▶ chaînes de caractères  
(par exemple, le « nom »)
- ▶ ...

## Types de base et types évolués

- ▶ Toute variable de type de base stocke directement **une valeur** :



- ▶ Toute variable de type évolué, comme les tableaux ou les chaînes de caractères (`String`) que vous allez voir dans ce cours, stocke **une référence** (adresse) vers une valeur :



## Types de base et types évolués (2)

- ▶ Toute variable de type de base stocke directement **une valeur**
- ▶ Toute variable de type évolué stocke **une référence** vers une valeur
- ☞ **Attention** : ceci a une très grande incidence sur la sémantique des opérateurs = et == en Java !
- ☞ Cela a aussi une incidence sur l'affichage
  - ▶ `double x = 13.5` veut dire « *J'affecte à x la valeur 13.5* »
  - ▶ `String s = "coucou"` veut dire « *J'affecte à s une référence à la chaîne de caractères coucou* »

En clair, si `v1` et `v2` sont de type évolué :

- ▶ `v1 = v2` affecte l'adresse de `v2` à la variable `v1`
- ▶ `v1 == v2` compare l'adresse de `v2` avec celle de `v1`
- ▶ `System.out.println(v1);` affiche l'adresse de `v1` (dans le cas général)

Nous y reviendrons ...

## Petit exemple introductif

Supposons que l'on souhaite écrire un programme de jeu à plusieurs joueurs.

Score	Écart à la moyenne
1000	-1860
1500	-1360
2490	-370
6450	3590
...	...

Commençons modestement par... deux joueurs.

## Solution avec les moyens actuels

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

// Lecture des données et calculs
System.out.println ("Score Joueur 1:");
int score1 = keyb.nextInt();
System.out.println ("Score Joueur 2:");
int score2 = keyb.nextInt();
// Calcul de la moyenne
double moyenne = (score1 + score2);
moyenne /= 2;
// Affichages
System.out.println("Score      Ecart Moyenne");
System.out.println(score1 + " " + (score1 - moyenne));
System.out.println(score2 + " " + (score2 - moyenne));
...
```

Comment passer à plus de joueurs ?

☞ utiliser plus de variables

## Solution avec les moyens actuels (2)

```
System.out.println ("Score Joueur 1:");
int score1 = keyb.nextInt();
System.out.println ("Score Joueur 2:");
int score2 = keyb.nextInt();
System.out.println ("Score Joueur 3:");
int score3 = keyb.nextInt();
System.out.println ("Score Joueur 4:");
int score4 = keyb.nextInt();
System.out.println ("Score Joueur 5:");
int score5 = keyb.nextInt();

// calcul de la moyenne
double moyenne = (score1 + score2 + score3 + score4 + score5);
moyenne /= 5;
```

Comment faire les affichages ?

## Solution avec les moyens actuels (3)

```
System.out.println ("Score Joueur 1:");
int score1 = keyb.nextInt();
System.out.println ("Score Joueur 2:");
int score2 = keyb.nextInt();
...
System.out.println ("Score Joueur 5:");
int score5= keyb.nextInt();

// calcul de la moyenne
double moyenne = (score1 + score2 + score3 + score4 + score5);
moyenne /= 5;
```

```
// Affichages
System.out.println("Score          Ecart Moyenne");
for(int i = 1; i <= 5; ++i) {
    System.out.println(scorei + "      " + scorei - moyenne);
}
```

## Solution avec les moyens actuels : limites

Mais

1. comment l'écrire (`scorei` n'est pas correct) ?
2. comment faire si on veut considérer 100, 1000... joueurs ?
3. comment faire si le nombre de joueurs n'est pas connu au départ ?

🔑 Solution : les **tableaux**

## Solution avec tableau

```
System.out.print ("Donnez le nombre de joueurs:");
int n = keyb.nextInt();
if (n > 0) {
    double moyenne = 0;
    int scores [] = new int[n];

    // Lecture des scores
    for (int i = 0; i < n; ++i) {
        System.out.println ("Score Joueur " + i + " :");
        scores[i] = keyb.nextInt();
        moyenne += scores[i];
    }
    moyenne /= n; // calcul de la moyenne

    // Affichages
    System.out.println(" Score " + " Ecart Moyenne");
    for (int i = 0; i < n ; ++i) {
        System.out.println(scores[i] + " " + (scores[i] - moyenne));
    }
}
```

## Les tableaux

Un **tableau** est une collection de valeurs *homogènes*, c'est-à-dire constitué d'éléments qui sont tous du **même type**.

Exemple : Tableau `scores` contenant 4 `int`

1000	1500	2490	6450
scores[0]	scores[1]	scores[2]	scores[3]

On utilise donc les tableaux lorsque *plusieurs* variables de *même* type doivent être *stockées*/mémorisées.

On pourra définir des tableaux d'`int`, de `double`, de `bool`, ...  
... mais aussi de n'importe quel autre type à disposition  
par exemple des tableaux de tableaux.

## Les différentes sortes de tableaux

Il existe en général quatre sortes de tableaux :

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	1.	2.
	non	3.	4.

Remarques :

- ▶ avec le premier type de tableau (1.), on peut faire tous les autres  
☞ les autres permettent des *optimisations*
- ▶ pratiquement aucun langage de programmation n'offre les 4 variantes

## Les tableaux en Java

En Java, on utilise :

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	ArrayList	ArrayList
	non	tableaux de taille fixe	tableaux de taille fixe

Dans un premier temps, nous allons nous intéresser aux

**tableaux de taille fixe**

dont la taille, en Java, peut se décider avant ou pendant l'exécution, mais qui une fois choisie ne peut plus varier pendant le déroulement du programme.

Viendront, lors du prochain cours, les tableaux dynamiques, dont la taille peut varier pendant l'exécution du programme.

## Déclaration d'un tableau de taille fixe

Syntaxe générale : Type des éléments + crochets [].

```
int[] scores;
```

Note : Java autorise la syntaxe équivalente suivante :

```
int scores[];
```

Les crochets indiquent que la variable peut contenir plusieurs éléments du type spécifié

Il existe **deux techniques pour initialiser** les éléments :

1. Dans l'instruction de déclaration
2. Dans des instructions séparées

## Initialisation d'un tableau de taille fixe (1)

Si l'on connaît les valeurs de tous les éléments lors de la déclaration du tableau

☞ une seule instruction de **déclaration-initialisation**

1. Déclarer le type du tableau
2. Indiquer les éléments entre accolades
3. Séparer les éléments par des virgules

Exemple : avec des **littéraux** :

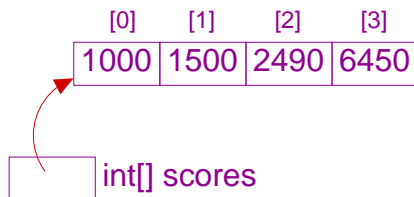
```
int[] scores = {1000, 1500, 2490, 6450};
```

Exemple : avec des **expressions** :

```
int[] scores = {(2500 * 10), (4200 * 10)};
```

## Situation en mémoire

**Important** : Un tableau n'est pas de type de base, il est donc manipulé via une **référence** !



On dit que la variable **scores** **référence** (ou pointe vers) un tableau de **int**.

La variable **scores** contient une adresse : l'emplacement du tableau en mémoire !

## Initialisation d'un tableau de taille fixe (2)

Dans le cas général, on ne connaît pas les valeurs de tous les éléments lors de la déclaration du tableau

On utilise alors plusieurs instructions pour **déclarer et initialiser** :

1. Déclarer le type du tableau
2. Construire le tableau avec :  
`new type [ taille ]`
3. remplir le tableau **élément par élément**

La **déclaration-construction** d'un tableau peut se faire avec :

► deux instructions distinctes :

```
int[] scores;           // déclaration
scores = new int[2];    // construction
```

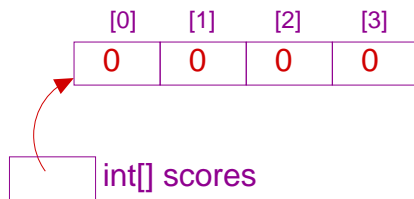
► une seule instruction :

```
int[] scores = new int[2]; // déclaration-
                          // -construction
```

## Valeurs par défaut

Chaque élément d'un tableau reçoit une **valeur par défaut** lors de la construction avec `new`

int	0
double	0.0
boolean	false
char	'\u0000'
(objet quelconque)	(null)

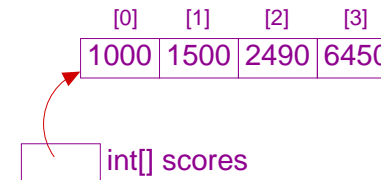


## Initialisation d'un tableau de taille fixe (3)

Une fois le tableau déclaré et construit, il faut le **remplir élément par élément** :

```
int[] scores = new int[4];
scores[0] = 1000;
scores[1] = 1500;
scores[2] = 2490;
scores[3] = 6450;
```

Situation en mémoire : Comme pour le 1<sup>er</sup> exemple d'initialisation




## Accès direct aux éléments d'un tableau

Le  $i+1^{\text{ème}}$  élément d'un tableau `tab` est accessible au moyen de l'indexation : `tab[i]`

**Attention !** Les indices correspondant aux éléments d'un tableau de taille `T` **varient entre 0 et T-1**

Le 1<sup>er</sup> élément d'un tableau `tab` précédemment déclaré est donc `tab[0]` et son 10<sup>e</sup> élément est `tab[9]`

**Attention !** En cas de débordement une exception est lancée par le programme  situation d'erreur provoquant l'arrêt du programme si on ne la traite pas (la gestion des exceptions n'est pas présentée dans ce cours d'introduction)

Il est impératif que l'élément auquel vous voulez accéder **existe** effectivement !

## Affichage d'un tableaux de taille fixe

Le code suivant :

```
double[] t1 = {1.1, 2.2, 3.4};  
System.out.println(t1);
```

affiche la référence au tableau `t1`, donc une adresse.

Si l'on veut faire afficher les entrées du tableau référencé par `t1`, il faut prévoir une boucle (itération) !

## Accès aux éléments d'un tableau (1)

Très souvent, on voudra accéder aux éléments d'un tableau en effectuant une *itération* sur ce tableau.

Il existe en fait au moins *trois* façons d'itérer sur un tableau :

- ▶ avec les itérations sur ensemble de valeurs

```
for(Type element : tableau)  
    // Type est le type des éléments du tableau
```

- ▶ avec une itération `for` « classique » :

```
for(int i=0; i < TAILLE; ++i)  
    // TAILLE ?? voir plus loin
```

- ▶ avec des itérateurs (non présenté dans ce cours)

## Accès aux éléments d'un tableau (2)

Attention, les itérations sur ensemble de valeurs :

```
for(Type element : tableau)
```

est très simple et élégant mais :

- ▶ ne permet pas modifier le contenu du tableau
- ▶ ne permet d'itérer que sur un seul tableau à la fois : il n'est pas possible de traverser en une passe deux tableaux pour les comparer par exemple
- ▶ ne permet l'accès qu'à un seul élément : on ne peut pas par exemple comparer un élément du tableau et son suivant
- ▶ itère d'un pas en avant seulement.

## Nombre d'éléments d'un tableau

Pour connaître la **taille d'un tableau** :

- ▶ `nomTableau.length`

Exemple :

```
int[] scores = {1000, 1500, 2490, 6450};  
System.out.println(scores.length); // 4  
boolean[] bs = {true, false};  
System.out.println(bs.length); // 2
```

**Attention !** `length` donne le **nombre possible** d'éléments. Le remplissage effectif du tableau n'a pas d'importance !

Exemple :

```
int[] scores = new int[2];  
System.out.println(scores.length); // 2
```

## Erreurs courantes avec les tableaux

1. Problème d'indice
2. Accès avant la construction du tableau
- (3.) Accès à un élément non initialisé (objets, valeur `null`)
- (4.) Confusion syntaxique tableau/objet

Les deux derniers types d'erreurs seront exposés après introduction de la notion d'objet

## Erreur : problème d'indice

### Attention !

- ▶ L'indice est toujours un `int`
- ▶ Il faut **respecter les bornes** du tableau :
  - ☞ Toujours énumération de `[0]` à `[T-1]` (où `T` est la taille du tableau)

### Exemples :

```
int[] entiers = new int[250];
entiers[1.0] = 1; // erreur type
entiers[-13] = 2; // erreur borne
entiers[250] = 4; // erreur borne
```

## Erreur : accès avant construction

Il est impossible en Java d'accéder à un élément si le tableau n'a pas encore été construit.

La construction se fait :

- ▶ Soit en indiquant les valeurs directement dans l'instruction de déclaration
- ▶ Soit en spécifiant la taille avec `new type[taille]`

### Exemple :

```
int[] entiers1 = {1, 2, 3}; // Déclaration-initialisation
entiers1[0] = 4;           // OK
int[] entiers2;           // Déclaration
entiers2[0] = 4;           // Erreur !
```

## Erreur : accès avant construction

Il est impossible en Java d'accéder à un élément si le tableau n'a pas encore été construit.

La construction se fait :

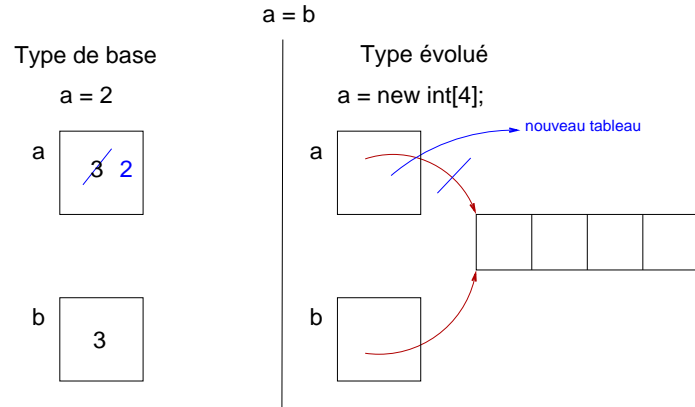
- ▶ Soit en indiquant les valeurs directement dans l'instruction de déclaration
- ▶ Soit en spécifiant la taille avec `new type[taille]`

### Exemple :

```
int[] entiers1 = {1, 2, 3}; // Déclaration-initialisation
entiers1[0] = 4;           // OK
int[] entiers2;           // Déclaration
entiers2 = new int[10];    // Initialisation
entiers2[0] = 4;
```



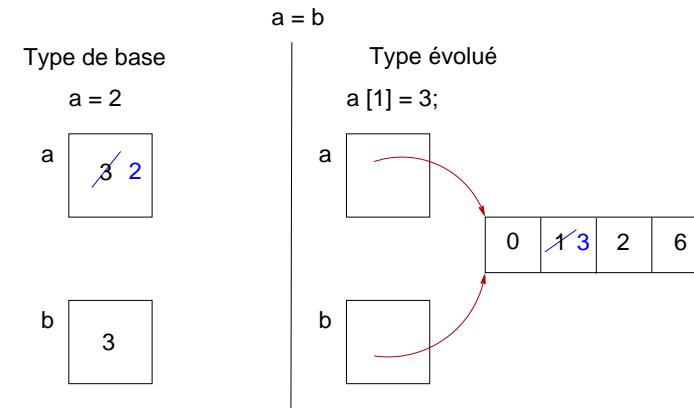
## Types de base / Types évolué (rappel)



☞ Type de base : modifier **a** ne modifie pas **b**

☞ Type évolué : modifier la référence **a** ne modifie pas la référence **b**

## Types de base / Types évolué (rappel)



☞ Type évolué : modifier (l'objet référencé par) **a** modifie (l'objet référencé par) **b**

## Tableaux : sémantique de l'opérateur =

```
// Les tableaux a et b pointent vers deux emplacements
// différents en memoire
int[] a = new int[10]; // tableau de 10 entiers
int[] b = new int[10]; // tableau de 10 entiers

for (int i = 0; i < a.length; ++i) {
    a[i] = i; // remplissage du tableau pointé par a
}
b = a; // opérateur = (affectation)
System.out.println("a[2] vaut " + a[2] + " et b[2] vaut " + b[2]);
a[2] = 42;
System.out.println("a[2] vaut " + a[2] + " et b[2] vaut " + b[2]);
```

ce qui affiche :

a[2] vaut 2 et b[2] vaut 2

a[2] vaut 42 et b[2] vaut 42

Les deux tableaux **a** et **b**, après l'affectation **b=a**; pointent vers le même emplacement mémoire ⇒ En changeant **a[2]** on change alors implicitement **b[2]** et inversement !

Le tableau créé pour **b** : `int[] b = new int[10];` n'est donc jamais rempli ni utilisé !

## Tableaux : utilisation (rare) de l'opérateur =

A moins de vouloir deux noms de variables pour le même tableau, il n'y a pas d'intérêt à vouloir assigner un tableau un à autre

☞ l'utilisation de l'opérateur **=** pour les tableaux est donc rare !

Pour avoir deux tableaux distincts **a** et **b** qui ont les mêmes valeurs (c'est-à-dire faire une copie de **a** dans **b**) il aurait fallu utiliser :

```
for(int i = 0; i < a.length; ++i) {
    b[i] = a[i];
}
```

**Attention** : il faut que `b.length ≥ a.length` !

## Tableaux : sémantique de l'opérateur == (1)

L'opérateur `a == b` teste si les variables `a` et `b` référencent le même emplacement mémoire.

⇒ ce qui est donc le cas lors de l'affectation `b = a;`

L'opérateur `a == b` **ne teste pas** l'égalité des valeurs contenues dans les tableaux pointés par `a` et `b` !

## Tableaux : sémantique de l'opérateur == (2)

Pour vérifier l'égalité de contenu des tableaux, il faut écrire explicitement les tests :

```
if (a == null || b == null || a.length != b.length) {
    System.out.println("contenus différents ou nuls");
}
else {
    int i = 0;
    while(i < a.length && (a[i] == b[i])) {
        ++i;
    }
    if (i >= a.length) {
        System.out.println("contenus identiques");
    }
    else {
        System.out.println("contenus différents")
    }
}
```

## Quelques exemples de manipulation de tableaux

Soit un tableau déclaré par :

```
double[] tab = new double[10];
```

Affichage du tableau :

- ▶ si l'on n'a pas besoin d'expliciter les indices :

```
System.out.print("Le tableau contient : ");
for(double element : tab) {
    System.out.print(element + " ");
}
System.out.println();
```

- ▶ si l'on veut expliciter les indices :

```
for(int i = 0; i < tab.length; ++i) {
    System.out.println("L'élément " + i + " vaut " + tab[i]);
}
```

## Quelques exemples de manipulation de tableaux

Saisie au clavier des éléments du tableau :

- ▶ Il est toujours nécessaire d'expliciter les indices :

```
for(int i = 0; i < tab.length; ++i) {
    System.out.println("Entrez l'élément " + i + " :");
    tab[i] = scanner.nextDouble();
}
```

## Tableaux à plusieurs dimensions

Comment déclarer un tableau à plusieurs dimensions ?

☞ On ajoute simplement un niveau de `[]` de plus :

C'est en fait un tableau de tableaux...

Exemples :

```
double[] [] statistiques =  
    new double[nbCantons][nbCommunes];  
  
int[] [] scores =  
    new int[nbJoueurs][nbParties];
```

`scores[i]` est un tableau de `nbParties` entiers

☞ `scores` est bien un tableau de tableaux.

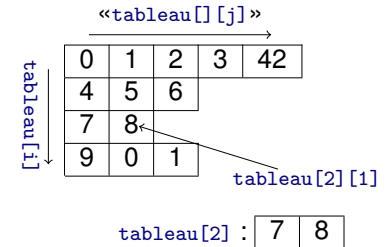
En faisant une analogie avec les mathématiques, un tableau à une dimension représente donc un **vecteur**, un tableau à deux dimensions une **matrice** et un tableau de plus de deux dimensions un **tenseur**.

## Tableaux à plusieurs dimensions

Les tableaux multidimensionnels peuvent également être initialisés lors de leur déclaration. Il faut bien sûr spécifier autant de valeurs que les dimensions et ceci pour chacune des dimensions.

Exemple :

```
int[] [] tableau = {  
    { 0, 1, 2, 3, 42 },  
    { 4, 5, 6 },  
    { 7, 8 },  
    { 9, 0, 1 }  
};
```



## Déclaration-initialisation (1)

Cas 1 : On connaît tous les éléments lors de la déclaration

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

Accès aux éléments de la 1<sup>re</sup> dimension :

- Type `int[]`
- `y[0]`, `y[1]` et `y[2]`

Accès aux éléments de la 2<sup>e</sup> dimension :

- Type `int`
- `y[0][0]` `y[0][1]` (1<sup>er</sup> tableau)
- `y[1][0]` `y[1][1]` (2<sup>e</sup> tableau)
- `y[2][0]` `y[2][1]` (3<sup>e</sup> tableau)

## Déclaration-initialisation (2)

Cas 2 : On ne connaît pas tous les éléments lors de la déclaration

```
int[] [] y = new int[3][2];  
  
// remplissage à la main  
  
y[0][0] = 1;  
y[0][1] = 2;  
  
y[1][0] = 3;  
y[1][1] = 4;  
  
y[2][0] = 5;  
y[2][1] = 6;
```

## Parcours

Le moyen le plus naturel de parcourir un tableau multidimensionnel consiste à utiliser des boucles `for` imbriquées :

1. 1<sup>re</sup> boucle : fait varier le 1<sup>er</sup> indice
2. 2<sup>e</sup> boucle : fait varier le 2<sup>e</sup> indice

Exemple :

```
for(int i = 0; i < y.length; ++i) {  
    for(int j = 0; j < y[i].length; ++j) {  
        System.out.println(y[i][j]);  
    }  
}
```

## Parcours

Le moyen le plus naturel de parcourir un tableau multidimensionnel consiste à utiliser des boucles `for` imbriquées :

1. 1<sup>re</sup> boucle : fait varier le 1<sup>er</sup> indice
2. 2<sup>e</sup> boucle : fait varier le 2<sup>e</sup> indice

Exemple : Variante (tous les éléments de la 1<sup>re</sup> dimension ayant la même taille)

```
System.out.println(y[0].length); // 2  
System.out.println(y[1].length); // 2  
System.out.println(y[2].length); // 2  
for (int i = 0; i < y.length; ++i)  
    for (int j = 0; j < y[0].length; ++j)  
        System.out.println (y[i][j]);
```