



LU2IN002 TABLEAUX

Vincent Guigue & Christophe Marsala



Tableau =

La **structure de base** de la programmation impérative: disponible sur **les types de base et sur les objets**.

1 Tableau à taille fixe

- + Economie mémoire
- + Rapidité d'accès
- Peu flexible (taille fixe !)

2 Tableau à taille variable

- Gourmand en mémoire
- (Un peu) moins rapide
- + Très flexible

[Taille fixe] Syntaxe: un objet *presque* comme les autres

- Déclaration : `type [] nomVariable`
- Instanciation : `nomVariable = new type [taille]`
- Accès à la case i (lecture ou écriture) : `nomVariable[i]`
- accès à la longueur du tableau : `nomVariable.length`

```

1 public class TableauA {
2     public static void main(String[] argv) {
3         int[] tableau; // declaration
4
5         tableau = new int[2]; // instanciation
6         tableau[0] = 1; // utilisation (écriture)
7         tableau[1] = 4;
8
9         int i = tableau[0]; // utilisation en lecture
10        // acces a la longueur du tableau
11        System.out.println("Longueur: "+tableau.length);
12    }
13 }

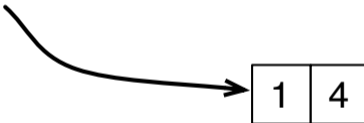
```

- `tableau` est une variable de type `int[]` (ie tableau d'entiers)
- `tableau[i]` : chaque case de tableau est de type `int`

```
1 int [] tableau = new int [2];  
2 tableau[0] = 1;  
3 tableau[1] = 4;  
4 // comme avec les objets...  
5 int [] tab2 = tableau;  
6 // pas d'instance suppl.
```

tableau =
création d'un **ensemble de variables**...
...facilement accessibles dans les boucles.

int[] tableau



Attention: bien différencier variables et instances...

- instantiation d'un tableau = création de **variables**
- ⇒ passage aux objets (un peu) piégeux

Soit la classe `Point` (vue dans les cours précédent)

Déclaration d'une variable `tabP` de type
`Point[]` (tableau de points)

```
1 Point[] tabP;
```

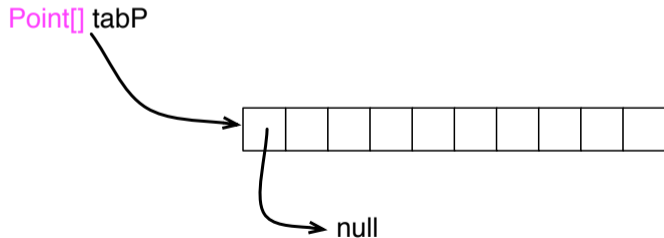
 Le tableau n'existe pas ! (il n'est **pas** instancié)

`Point[] tabP`

La variable `tabP` référence un tableau de 10 cases

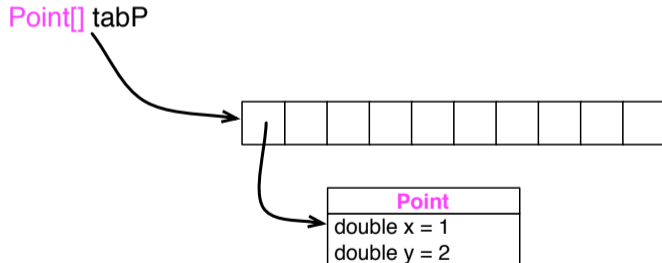
⚠ **10 cases = 10 variables...**
... **0 instance** de **Point** !

```
1 Point [] tabP;  
2 tabP = new Point [10];
```

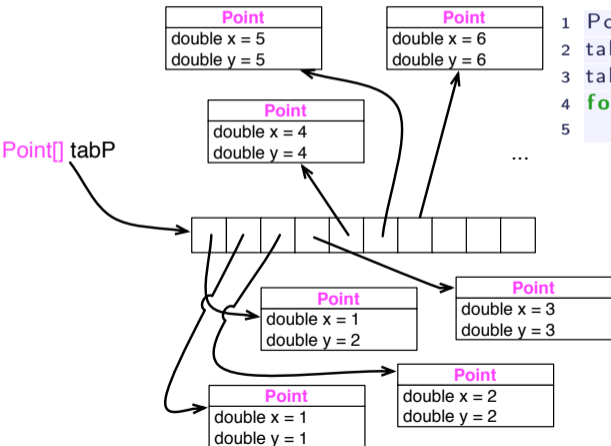


Chaque case (=variable) doit être instanciée

```
1 Point [] tabP;  
2 tabP    = new Point [10];  
3 tabP [0] = new Point (1,2);
```

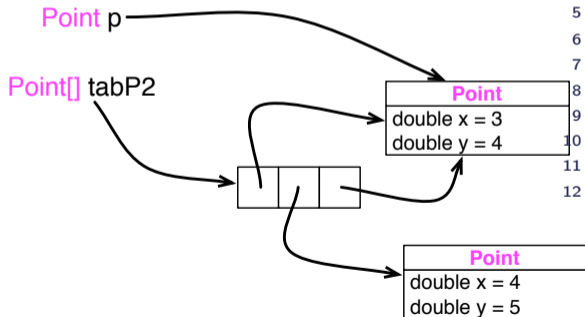


Usage parfait en combinaison des boucles:



```
1 Point [] tabP;  
2 tabP = new Point [10];  
3 tabP [0] = new Point (1,2);  
4 for (int i=1; i<tabP.length; i++)  
5   tabP [i] = new Point (i,i);
```


Les cases se comportent vraiment comme des variables: on peut jouer avec les références



```
1 Point [] tabP;  
2 tabP = new Point [10];  
3 tabP [0] = new Point (1,2);  
4 for (int i=1; i<tabP.length; i++)  
5     tabP [i] = new Point (i,i);  
6 // second tableau  
7 // + jeu de references  
8 Point [] tabP2 = new Point [3];  
9 Point p = new Point (3,4);  
10 tabP2 [0] = p;  
11 tabP2 [1] = new Point (4,5);  
12 tabP2 [2] = tabP2 [0];
```

Création Syntaxe simplifiée : {value,value,...}

 Ne marche que sur la ligne de déclaration

```
1 boolean [] tableau={true , false , true };
```

Création Syntaxe simplifiée : {value,value,...}

⚠ Ne marche que sur la ligne de déclaration

```
1 boolean [] tableau={true , false , true };
```

Création Syntaxe intermédiaire (marche partout):

```
new type [] {value,value,...}
```

```
1 boolean [] tableau;  
2 tableau = new boolean []{ true , false , true };
```

Boucle Parcours des éléments du tableau (sans référence d'indice):

```
for(type var : nomTableau) ...
```

`var` prend successivement toutes les valeurs des éléments du tableau

```
1 for(boolean b: tableau) // affichage de tous les elements
2   System.out.println(b);
```



Pas de référence aux indices: usage possible, ou pas, en fonction des algorithmes

Code robuste = pas de duplication de l'information

Attention aux conditions de fin de boucles

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

Code robuste = pas de duplication de l'information

Attention aux conditions de fin de boucles

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

Besoin de faire une boucle sur le tableau...

```
2 for (int i=0; i<5; i++) // FAUX dans le cadre de lu2in002
3     ... tab[i] ...
```

Code robuste = pas de duplication de l'information

Attention aux conditions de fin de boucles

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

Besoin de faire une boucle sur le tableau...

```
2 for(int i=0; i<5; i++) // FAUX dans le cadre de lu2in002
3   ... tab[i] ...
```

⇒ Modifier le tableau = bug dans le programme !

length

La taille du tableau `tab` est définie lors de la **création** (implicitement ou explicitement).
Utiliser `tab.length` pour y faire référence

```
4 // OK: modifier le tableau = modifier la boucle !
5 for(int i=0; i<tab.length; i++)
6   ... tab[i] ...
```

Usage dans 2 cas (imbriqués):

- **Taille finale inconnue** lorsque l'on commence à utiliser le tableau
(e.g. lecture d'un fichier...)
- **Taille variable** en cours d'utilisation (e.g. pile d'objets à traiter de taille variable)

■ Objet JAVA à déclarer avant utilisation:


```
1 import java.util.ArrayList; // en entete
```

■ Syntaxe objet classique + approche générique (hors prog.):

- la variable sera de type : `ArrayList<type>`
 - *type* est forcément un objet (\neq type de base): Integer, Double, Point...
- Même représentation mémoire que les tableaux de taille fixe

- Exemple sur un tableau de Point
- Méthodes principales: constructeur, add, get, remove, size
- Plus d'informations sur la javadoc (beaucoup de d'autres méthodes disponibles):
<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

```
1 // Construction comme un objet classique
2 ArrayList<Point> tabArr = new ArrayList<Point>();
3 tabArr.add(new Point(1,2)); // ajout
4 for(int i=0; i<9; i++) tabArr.add(new Point(i, i));
5
6 // accesseur sur le deuxieme element (index = 1)
7 Point p = tabArr.get(1);
8 tabArr.remove(0); // suppression du premier element
9
10 // accesseur sur la taille courante
11 System.out.println(tabArr.size());
```

 Tableau... \Rightarrow possibilité de sortir du tableau

- Cas classique:
 - Mélange entre taille n et dernier indice du tableau ($n - 1$)
 - Tentative d'accès à un index négatif
 - Erreur de boucle...
- Symptôme: **ArrayIndexOutOfBoundsException**
 - Echec lors de l'exécution du code (compilation OK)

```
1 Point [] tab = {new Point(), new Point()};  
2 System.out.println(tab[2]);
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2 at  
test.Point.main(Point.java:118)
```

- Attention aux **NullPointerException**: après instantiation d'un tableau, aucune instance n'est disponible:

```
1 Point [] tab = new Point[2];  
2 System.out.println(tab[0].getX()); //  $\Rightarrow$  NullPointerException
```

■ ArrayList

- Dans la classe même:

```
1 ArrayList<Double> arr = new ArrayList<Double>();
2 for(int i= 0; i<10; i++)
3     arr.add((double) i);
4 if(arr.contains(2.))
5     System.out.println("Trouve!");
```

- Dans la classe Collections

- Tris, min, max, mélange, renversement...

■ Tableau

- Dans la classe Arrays: recherche, copie, remplissage

```
1 int [] tab = {3, 5, 1, 8, 9};
2 System.out.println(Arrays.binarySearch(tab, 1));
```

Comment gérer les matrices?

⇒ Comme **des tableaux de tableaux**

■ Déclaration des variables : `type [] []`

```
1 int [][] matrice;
```

■ Instanciation

```
2 matrice = new int [2][3]; // 2 lignes , 3 colonnes
```

■ Usage

```
3 matrice [0][0] = 0; matrice [0][1] = 1; matrice [0][2] = 2;  
4 matrice [1][0] = 3; matrice [1][1] = 4; matrice [1][2] = 5;
```

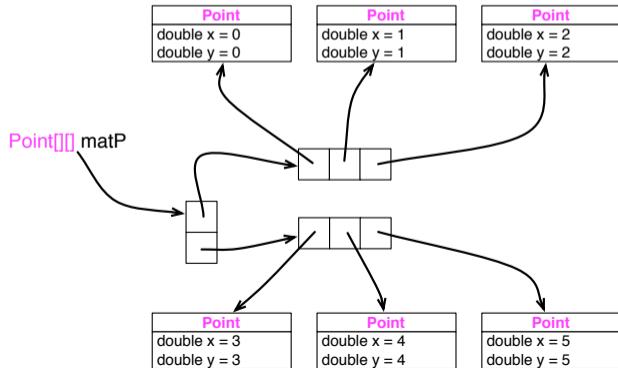
■ Syntaxe alternative d'instanciation/initialisation

```
6 int [][] matrice = {{0, 1, 2},{3, 4, 5}}
```

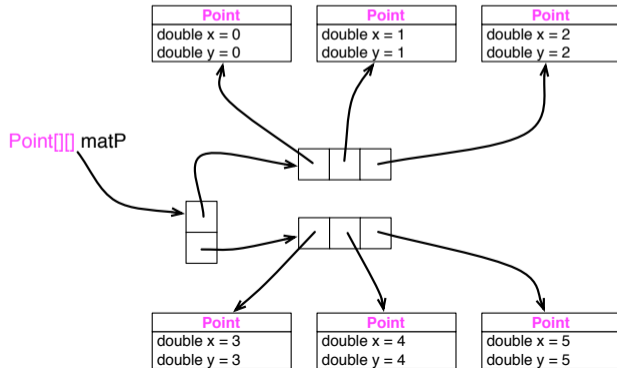
■ Accès aux dimensions:

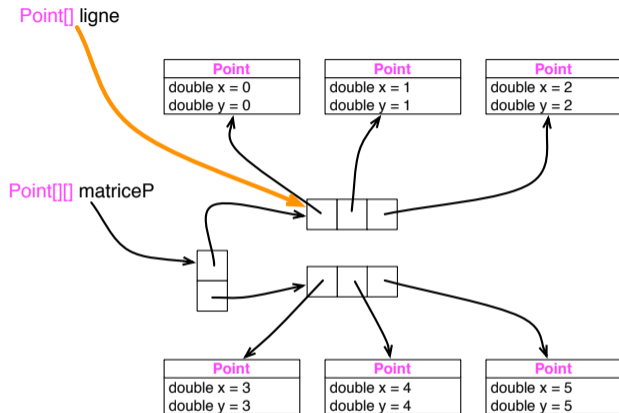
```
1 matrice.length // nb lignes  
2 matrice [0].length // nb de colonnes de la premiere ligne
```

```
1 Point [][] matriceP = new Point [2][3];
```



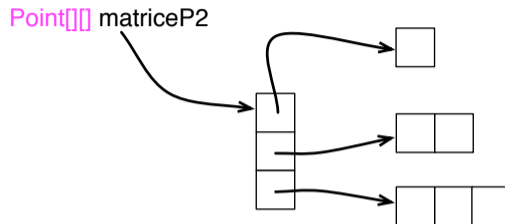
```
1 Point [][] matriceP = new Point [2][3];  
2 // est equivalent a :  
3 //     creation d'un tableau de tableau de taille 2  
4 Point [][] matriceP2 = new Point [2][];  
5 //     creation de 2 tableaux de 3 cases  
6 for (int i=0; i<matriceP2.length; i++)  
7     matriceP2[i] = new Point [3];
```





■ Possibilité de manipuler les lignes de la matrice de manière indépendante

```
1 Point [][] matriceP = new Point [2][3];  
2 Point [] ligne = matriceP [0];  
3 System.out.println(ligne [0]); // Affichage du premier point  
4 ligne [1] = new Point (6, 6);
```



■ Possibilité de manipuler les lignes de la matrice de manière indépendante

```
1 Point [][] matriceP = new Point [2][3];  
2 Point [] ligne = matriceP [0];  
3 System.out.println(ligne [0]); // Affichage du premier point  
4 ligne [1] = new Point (6, 6);
```

■ Construction de matrice triangulaire

```
1 Point [][] matriceP2 = new Point [3][];  
2 for(int i=0; i<matriceP2.length; i++)  
3     matriceP2 [i] = new Point [i+1];
```


■ Conversion sur les types de base: OK

```
1 double d = 2.4;  
2 int i = (int) d; // conversion explicite
```

■ Conversion sur les tableaux: KO, impossible

```
1 double[] tab = {2.5, 5, 8.};  
2 // aucune conversion possible  
3 // seule option: reconstruction complete:  
4 int[] tabInt = new int[tab.length];  
5 for(int i=0; i<tab.length; i++)  
6     tabInt[i] = (int) tab[i];
```

Même fonctionnement avec les tableaux ou les ArrayList

Tableau = ensemble de variables

Pas de flexibilité à ce niveau là...

... A voir avec l'héritage

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};  
2  int [] tab2 = {1, 2, 3, 4};  
3  int [] tab3 = tab1;  
4  
5  System.out.println(tab1 == tab2);
```

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};  
2  int [] tab2 = {1, 2, 3, 4};  
3  int [] tab3 = tab1;  
4  
5  System.out.println(tab1 == tab2); // false  
6  System.out.println(tab1 == tab3);
```

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};
2  int [] tab2 = {1, 2, 3, 4};
3  int [] tab3 = tab1;
4
5  System.out.println(tab1 == tab2); // false
6  System.out.println(tab1 == tab3); // true
7  System.out.println(tab1.length == tab2.length);
```

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};
2  int [] tab2 = {1, 2, 3, 4};
3  int [] tab3 = tab1;
4
5  System.out.println(tab1 == tab2); // false
6  System.out.println(tab1 == tab3); // true
7  System.out.println(tab1.length == tab2.length); // comp. entre int: true
8  System.out.println(tab1[0] == tab2[0]);
```

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};
2  int [] tab2 = {1, 2, 3, 4};
3  int [] tab3 = tab1;
4
5  System.out.println(tab1 == tab2); // false
6  System.out.println(tab1 == tab3); // true
7  System.out.println(tab1.length == tab2.length); // comp. entre int: true
8  System.out.println(tab1[0] == tab2[0]); // comp. entre int: true
9  System.out.println(tab1[0] == tab2[4]);
```

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};
2  int [] tab2 = {1, 2, 3, 4};
3  int [] tab3 = tab1;
4
5  System.out.println(tab1 == tab2); // false
6  System.out.println(tab1 == tab3); // true
7  System.out.println(tab1.length == tab2.length); // comp. entre int: true
8  System.out.println(tab1[0] == tab2[0]); // comp. entre int: true
9  System.out.println(tab1[0] == tab2[4]); // ERR exec IndexArrayOutOfBounds
10
11 tab3 = tab2;
```

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};
2  int [] tab2 = {1, 2, 3, 4};
3  int [] tab3 = tab1;
4
5  System.out.println(tab1 == tab2); // false
6  System.out.println(tab1 == tab3); // true
7  System.out.println(tab1.length == tab2.length); // comp. entre int: true
8  System.out.println(tab1[0] == tab2[0]); // comp. entre int: true
9  System.out.println(tab1[0] == tab2[4]); // ERR exec IndexArrayOutOfBounds
10
11  tab3 = tab2; // OK: representation memoire changee
12  // la variable tab3 reference le deuxieme tableau {1, 2, 3, 4}
13  tab3[0] = tab2[1];
```


Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};
2  int [] tab2 = {1, 2, 3, 4};
3  int [] tab3 = tab1;
4
5  System.out.println(tab1 == tab2); // false
6  System.out.println(tab1 == tab3); // true
7  System.out.println(tab1.length == tab2.length); // comp. entre int: true
8  System.out.println(tab1[0] == tab2[0]); // comp. entre int: true
9  System.out.println(tab1[0] == tab2[4]); // ERR exec IndexArrayOutOfBounds
10
11 tab3 = tab2; // OK: representation memoire changee
12 // la variable tab3 reference le deuxieme tableau {1, 2, 3, 4}
13 tab3[0] = tab2[1]; // OK: tab3 -> {2, 2, 3, 4}
14 tab3[1] = tab1;
```

Que penser des lignes de code suivantes:

- Est ce que ça compile?
- Est ce que ça s'exécute?
- Qu'est ce qui s'affiche? Quel est l'état de la mémoire?

```
1  int [] tab1 = {1, 2, 3, 4};
2  int [] tab2 = {1, 2, 3, 4};
3  int [] tab3 = tab1;
4
5  System.out.println(tab1 == tab2); // false
6  System.out.println(tab1 == tab3); // true
7  System.out.println(tab1.length == tab2.length); // comp. entre int: true
8  System.out.println(tab1[0] == tab2[0]); // comp. entre int: true
9  System.out.println(tab1[0] == tab2[4]); // ERR exec IndexArrayOutOfBounds
10
11 tab3 = tab2; // OK: representation memoire changee
12 // la variable tab3 reference le deuxieme tableau {1, 2, 3, 4}
13 tab3[0] = tab2[1]; // OK: tab3 -> {2, 2, 3, 4}
14 tab3[1] = tab1; // ERR compil int n'est pas compatible avec int[]
```

```
1 public boolean
2   egalite(int [] t1, int [] t2){
3
4   boolean b = true;
5
6   if(t1.length == t2.length){
7     for(int i=0; i<t1.length; i++){
8       if(t1[i] != t2[i])
9         b = false;
10    }
11  }
12  else
13    b = false;
14
15  return b;
16 }
```

```
1 public boolean
2   egalite(int [] t1, int [] t2){
3
4   boolean b = true;
5
6   if(t1.length == t2.length){
7     for(int i=0; i<t1.length; i++){
8       if(t1[i] != t2[i])
9         b = false;
10    }
11  }
12  else
13    b = false;
14
15  return b;
16 }
```

```
1 public boolean
2   egalite(int [] t1, int [] t2){
3
4   if(t1.length != t2.length)
5     return false;
6
7   for(int i=0; i<t1.length; i++)
8     if(t1[i] != t2[i])
9       return false;
10
11  return true;
12 }
```

`return`, `break`, `continue` permettent de réduire les imbrications de boucles et rendent le code plus lisible.

⇒ Utilisez-les !