

Utilisation de tableaux en Java

La syntaxe Java de définition d'un tableau

La syntaxe de base

Bien entendu, comme tout langage de programmation digne de ce nom, Java permet d'utiliser les tableaux. Un tableau est un ensemble d'éléments basé sur un même type : par exemple, un tableau d'entiers, un tableau de chaînes de caractères, ... Les éléments d'un tableau sont placés consécutivement en mémoire, du premier au dernier. Un tableau sera introduit par la syntaxe `[]` qui pourra être placée à la suite du type utilisé pour les éléments stockés dans ce tableau. En fait, ces deux éléments (type suivi des crochets) définissent votre « type tableau ». A la suite des crochets, devra suivre le nom de la variable, ainsi qu'une éventuelle liste d'initialisation.

```
1 int [] myArray1;
2 int [] myArray3 = { 10, 20, 30 };
```

Un tableau de 10 valeurs entières


Pour accéder à un élément du tableau, on utilisera le nom de la variable associé ainsi que son indice (une valeur numérique indiquant la position de l'élément dans le tableau) compris entre deux crochets (crochets ouvrant et crochet fermant).

Il est possible de demander la taille d'un tableau via l'attribut `length` : celui-ci est en lecture seule. Il ne sera donc plus possible de changer sa taille après son initialisation. Par contre, il sera possible de stocker dans la variable un nouveau tableau qui lui pourra avoir une nouvelle taille. En fait, un tableau est un type objet : il est donc géré par références (par pointeurs, si vous préférez).

Voici un exemple d'utilisation d'un tableau de 10 valeurs entières. Ici le tableau est défini via une liste d'initialisation (la séquence de valeurs comprises entre les deux accolades).

```
1 public class Demo {
2
3     public static void main( String [] args ) {
4
5         // Le tableau est typé int []
6         // Il est initialisé avec 10 valeurs (les multiples de 10).
7         int [] myArray = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
8
9         // Quelle est la taille du tableau myArray ?
10        System.out.println( "Size == " + myArray.length );
11
12        // On accède à certaines valeurs de ce tableau
13        System.out.println( myArray[0] ); // Affiche 10
14        System.out.println( myArray[5] ); // Affiche 60
15        System.out.println( myArray[9] ); // Affiche 100
16
17    }
18
19 }
```

Un tableau de 10 valeurs entières

 **ATTENTION** : en Java, les indices d'un tableau commencent à 0. Ainsi, si l'on utilise un tableau de 10 éléments, les indices de ce tableau iront de 0 à 9.

En fait, depuis notre premier exemple de code Java, nous avons introduit le concept de tableau. Effectivement, tout programme Java peut être lancé à partir de la ligne de commande et peut donc accepter un certain nombre d'arguments. La méthode statique `main` prend donc, en paramètre, en tableau de chaînes de caractères appelé `args` : il correspond à l'ensemble des arguments fournis sur la ligne de commande. Il est lui aussi indicé à partir de 0.

Attention aux bornes de votre tableau

Nous l'avons dit, dans un tableau, les éléments sont placés en séquence (les uns derrières les autres) et on y accède via un indice. De plus, ces indices doivent être basés à partir de 0. D'où la question suivante : que ce passe-t-il si l'on exécute le programme suivant ?


```
1 public class Demo {
2
3     public static void main( String [] args ) {
4
5         int [] myArray = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
6
7         // Quelle est la taille du tableau myArray ?
8         System.out.println( "Size == " + myArray.length );
9
10        // On imagine accéder à certaines valeurs de ce tableau.
11        // Mais dans les deux cas, on est hors bornes.
12        System.out.println( myArray[-1] );
13        System.out.println( myArray[10] );
14
15    }
16
17 }
```

Attention aux bornes de votre tableau

Voici le résultat produit par l'exécution de ce code : une exception (une erreur) est déclenchée.

```
$> javac Demo.java
$> java Demo
Size == 10
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
    at Demo.main(Demo.java:12)
$>
```

Ce qui est affiché sur la console se nomme la « call stack » (la pile des appels de méthodes). Cette « call stack » indique que l'exception déclenchée (de type `java.lang.ArrayIndexOutOfBoundsException`) a été produite par la ligne 12 du fichier `Demo.java` et plus précisément dans sa méthode `main`.

 **Note** : bien entendu, les exceptions doivent être évitées, tant que possible. Et si d'aventure des exceptions étaient produites par votre programme, elles devraient être interceptées et traitées. Nous reviendrons dans des futurs chapitres sur le traitement des exceptions.

Les différentes syntaxes possibles

Pour introduire une variable de type tableau il vous faut utiliser la syntaxe []. Néanmoins les crochets peuvent être placés à différents endroits. Le tableau ci-dessus vous propose quelques exemples de définition de tableaux Java. Pour information, en Java, un tableau est un objet, il est donc géré par référence (par un pointeur). Une variable typée tableau contiendra donc un pointeur : à défaut d'initialisation, elle aura la valeur `null` et devra être initialisée ultérieurement. L'initialisation d'un tableau peut s'effectuer soit via un appel à l'opérateur `new`, soit via une liste d'initialisation.

Définition de tableau	Explications complémentaires
<code>int [] tb;</code>	Définit un tableau d'entiers non encore initialisé.
<code>int [] tb = new int[10];</code>	Définit un tableau de dix entiers. Il n'est plus possible de changer la taille de ce tableau.
<code>int [] tb1, tb2;</code>	Comme les crochets sont placés à gauche des noms de variables ils s'appliquent à toutes ces variables. On est donc en train de définir deux tableaux d'entiers, tous deux non initialisés.
<code>int [] tb1 = new int[5], tb2 = new int[10];</code>	Définit un premier tableau de cinq entiers, puis un second de dix entiers.
<code>int tb[] = new int[10], value = 10;</code>	Définit un tableau de dix entiers ainsi qu'une variable (value) de type entier. Les crochets ne s'appliquent uniquement qu'à <code>tb</code> car ils sont placés à la droite du nom de la variable.
<code>String [] strValues = new String[5];</code>	Définit un tableau pouvant contenir jusqu'à cinq chaînes de caractères.
<code>String [] strValues = { "toto", "titi", "tata" };</code>	Définit un tableau de chaînes de caractères contenant les trois valeurs stockées entre les accolades : on parle de liste d'initialisation.

Parcourir un tableau

Il existe plusieurs manières de parcourir un tableau : je vous en propose deux. Dans tous les cas, des instructions de boucles, non encore étudiées, sont utilisées pour ces parcours. Je vous renvoie sur les chapitres relatifs à ces instructions pour de plus amples détails.

Parcours avec un *for* traditionnel

Pour parcourir un tableau, il est possible d'utiliser l'instruction `for`. Elle existe depuis la version 1.0 de Java. Ce `for` doit avoir trois expressions placées entre parenthèses et séparées par des caractères `;`. Ensuite le corps de la boucle est placé entre accolades (en tout cas, c'est fortement conseillé).

```
1 for ( int i=0; i<array.length; i++ ) {
2     System.out.println( array[ i ] );
3 }
```

L'instruction de boucle for

Revenons sur les trois expressions placées entre parenthèses et voici leurs significations.

- `int i=0` : permet de déclarer et d'initialiser la variable `i` qui servira d'index lors du parcours.
- `i<array.length` : c'est la condition de rebouclage. Un nouveau tour de boucle a lieu tant que cette expression renvoie `true` : Dit autrement, on boucle tant que `i` est strictement inférieur à la taille du tableau.
- `i++` : cette expression, exécutée à chaque fin de tour de boucle, permet d'incrémenter la variable `i` d'une unité.

Pour ce qui est du corps de la boucle (placé entre accolades), il se contente d'afficher la valeur stockée dans chaque cellule du tableau (la valeur à l'indice `i`). Voici le programme complet.

```
1 public class Demo {
2
3     public static void main( String [] args ) {
4
5         // Déclaration et initialisation du tableau
6         int [] array = { 10, 20, 30, 40, 50 };
7
8         // Parcours du tableau
9         for ( int i=0; i<array.length; i++ ) {
10             System.out.println( array[ i ] );
11         }
12     }
13 }
14
15 }
```

L'instruction de boucle for

Et voici les résultats produits par ce programme.

```
10
20
30
40
50
```

ATTENTION : les développeurs Java parlent fréquemment de l'instruction « for each ». Pour autant, il n'y a pas de mot clé `foreach` qui existe en Java. Le mot clé associé à cette instruction « for each » est `for`, mais avec une syntaxe légèrement différente de celle présentée précédemment.

L'intérêt d'un « for each », par rapport à un `for` traditionnel, est que vous n'avez pas à gérer l'indice de boucle. Du coup, la syntaxe est un petit peu plus simple, comme en atteste le programme suivant en mettant les deux syntaxes côte à côte pour comparaison.

```
1 public class Demo {
2
3     public static void main( String [] args ) {
4
5         // Déclaration et initialisation du tableau
6         int [] array = { 10, 20, 30, 40, 50 };
7
8         System.out.println( "---Parcours du tableau avec un for traditionnel---" );
9         for ( int i=0; i<array.length; i++ ) {
10             int value = array[i];
11             System.out.println( value );
12         }
13
14         System.out.println( "---Parcours du tableau avec un for each---" );
15         for ( int value : array ) {
16             System.out.println( value );
17         }
18     }
19 }
20
21 }
```

L'instruction de boucle « for each »

Ce qui diverge, entre les deux syntaxes, c'est surtout ce qui est placé entre les parenthèses. Au lieu d'y gérer un indice de parcours, on dit simplement qu'on veut extraire toutes les données du tableau mentionné à la droite du caractère `:`. Le parcours sera, bien entendu, réaliser en commençant de la première donnée et jusqu'à la dernière. A la gauche du caractère `:`, on spécifie le type (`int`) et le nom (`value`) de la variable qui va recevoir, une à une, ces données. Si `value` ne vous convient pas, vous pouvez nommer cette variable comme bon vous semble.

Encore une fois, voici les résultats produit par ce programme. Bien entendu, nous réalisons deux fois la même chose, avec des techniques différentes : donc les résultats produits sont les mêmes.

```
---Parcours du tableau avec un for traditionnel---
10
20
30
40
50
---Parcours du tableau avec un for each---
```

Parcours des arguments passés à votre exécutable à partir de la ligne de commande

Comme nous l'avons dit plus haut, le paramètre `args` de la méthode `main` est un tableau de chaînes de caractères. Ce tableau contient normalement les paramètres passés à l'application lors de son démarrage. L'exemple proposé ci-dessous parcourt tous les paramètres passés dans `args` et part du principe qu'ils contiennent des valeurs entières. Du coup il en fait la somme et affiche cette somme sur la console.

```
1 public class Adder {
2
3     public static void main( String [] args ) {
4
5         int accumulator = 0;
6
7         for( String param : args ) {
8             accumulator += Integer.parseInt( param );
9         }
10
11         System.out.println( accumulator );
12     }
13 }
14
15 }
```

Parcours du tableau args.

Et voici quelques exemples de lancement de ce programme avec diverses listes d'arguments.

```
$> javac Adder.java
$> java Adder
0
$> java Adder 10 20 30
60
$> java Adder 10 20 30 40 50
150
```

Si vous souhaitez lancer le programme avec Eclipse (sans démarrer de console), il est possible de fixer les valeurs des arguments passés à votre application. Pour ce faire, cliquez avec le bouton droit de la souris en plein milieu du code de la classe `Adder.java`. Un menu contextuel doit apparaître : sélectionnez-y « Run As » puis « Run Configurations... ».



La boîte de dialogue présentée ci-dessous doit apparaître. Dans l'onglet « Main », vérifiez bien que la classe `Adder` soit sélectionnée comme étant la classe de démarrage, puis dans l'onglet « Arguments » renseignez les valeurs souhaitées au niveau de « Program Arguments ». Puis cliquez sur le bouton « Run » situé en bas et à droite de la boîte de dialogue. Le résultat devrait apparaître dans la console.

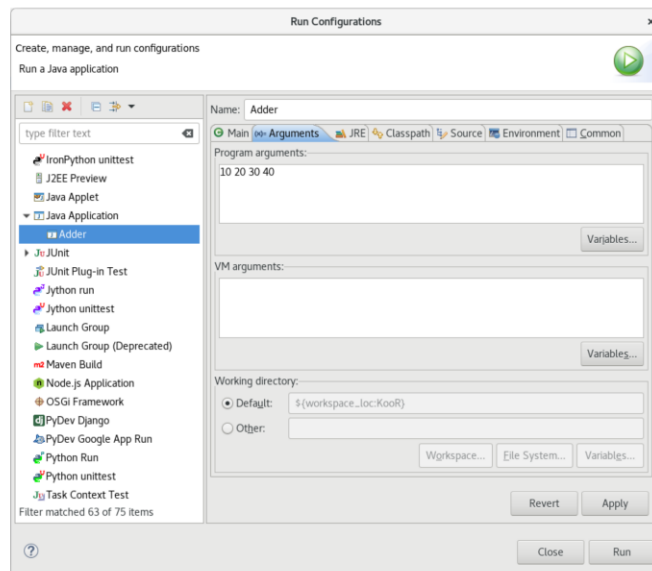


Tableau de tableaux (ou tableau multidimensionnels)

Imaginons que vous cherchiez à développer un jeu d'échec. Ce jeu se joue sur un plateau de 8 cases horizontalement sur 8 cases verticalement. Il y a donc 64 cellules dans lesquelles peuvent être placées les différentes pièces de jeu. Du coup, comment stocker, informatiquement parlant, l'état de l'échiquier en mémoire ? Et bien, nous allons certainement utiliser les tableaux !

Mais comment gérer les lignes et les colonnes de notre échiquier ? C'est là qu'intervient la notion de tableau multidimensionnel et plus précisément de tableau à deux dimensions dans notre cas. Regarder le code suivant : il créé le tableau associé à notre jeu d'échec, puis affiche le contenu de ce tableau à l'écran et enfin tente de déplacer une pièce.

```
1 public class ChessGame {
2
3     public static void displayBoard( String [][] board ) {
4         for ( int line=0; line<board.length; line++ ) {
5             for ( int column=0; column<board[line].length; column++ ) {
6                 System.out.print( board[line][column] + " " );
7             }
8             System.out.println();
9         }
10    }
11
12
13    public static void main( String [] args ) {
14
15        // Définition de notre échiquier.
16        // Conventions : TN=Tour Noire, CN=Cavalier Noir,
17        //                FN=Fou Noir, QN=Reine (Queen) Noire,
18        //                KN=Roi (King) Noir, PN=Pion Noir
19        // Idem pour les pièces blanches PB, TB, CB, ...
20        //                VD=ViDe
21        String [][] board = {
22            { "TN", "CN", "FN", "QN", "KN", "FN", "CN", "TN" },
23            { "FN", "PN", "PN", "PN", "PN", "PN", "PN", "PN" },
24            { "VD", "VD", "VD", "VD", "VD", "VD", "VD", "VD" },
25            { "VD", "VD", "VD", "VD", "VD", "VD", "VD", "VD" },
26            { "VD", "VD", "VD", "VD", "VD", "VD", "VD", "VD" },
27            { "VD", "VD", "VD", "VD", "VD", "VD", "VD", "VD" },
28            { "PB", "PB", "PB", "PB", "PB", "PB", "PB", "PB" },
29            { "TB", "CB", "PB", "QB", "KB", "PB", "CB", "TB" }
30        };
31
32        // Affichage de l'échiquier sur l'écran
33        displayBoard( board );
34        System.out.println( "-----" );
35
36        // On tente d'avancer le pion blanc devant le roi blanc
37        // d'une case en avant.
38        board[5][4] = "PB"; // 6ième ligne / 5ième colonne
39        board[6][4] = "VD"; // 7ième ligne / 5ième colonne
40
41        // Affichage de l'échiquier sur l'écran
42        displayBoard( board );
43    }
44
45
46 }
```

Implémentation d'un échiquier en mémoire

Quelques explications sont peut-être nécessaires. Le programme au niveau du `main` en ligne 13. La première chose qu'on y fait, c'est de déclarer une variable `board` : elle est de type `String [][]`, donc on parle bien d'un tableau de tableaux de chaînes de caractères. Le tableau de premier niveau va contenir des lignes. Chacune de ces lignes est donc bien un tableau de chaînes de caractères.

Comme indiqué dans les commentaires au-dessus de la déclaration de la variable `board`, les chaînes de caractères correspondent à l'état de la case considérée. La chaîne de caractère `"VD"` représente une case vide. Toutes les chaînes se terminant par `N` sont des pièces noires alors que celles se terminant par un `B` sont des pièces blanches. Et la première lettre d'une pièce représente sa nature : `P`=pion, `T`=tour, `C`=cavalier, `F`=fou, `K`=roi (king) et `Q`=reine (Queen).

Ensuite, on demande à afficher le tableau. Mais comme, nous allons devoir afficher le tableau deux fois, j'ai préféré déplacer le code d'affichage dans une méthode statique (en gros, une procédure) que je pourrais ensuite appeler autant de fois que nécessaire. Cette méthode statique se nomme `displayBoard` et elle débute en ligne 3 de l'exemple de code. Comme nous avons à tableau de tableau, deux boucles `for` sont nécessaire au parcours de toutes les cases de l'échiquier. La première, en ligne 4, permet de parcourir toutes les lignes : elle déclare un variable `line` qui correspond à l'indice de la ligne en cours de traitement. La seconde boucle, en ligne 5, permet le parcours d'une ligne : elle déclare aussi une variable nommée `column` qui correspond à l'indice de chaque cellule du tableau (de la ligne). La ligne 6 réalise l'affichage de chaque cellule. En fin de ligne, lors de l'affichage de l'échiquier, on rajoute un retour à la ligne pour que l'affichage soit parfait : c'est ce qui est réalisé en ligne 8 du code proposé.

Revenons au `main`, donc en ligne 33, on invoque `displayBoard` en lui passant en paramètre le tableau. Petit rappel, un tableau Java est un type objet, c'est donc une référence (un pointeur, si vous préférez) qui sera passé à la méthode. Il en résulte qu'il n'y a pas de clonage du tableau et donc, si par erreur, vous modifiez l'état du tableau dans la méthode d'affichage, alors cette modification sera aussi visible dans le `main`.

Ensuite, les deux lignes 38 et 39, effectuent un déplacement du pion situé devant le roi blanc. Et on finit, en ligne 42 par réafficher l'échiquier. Voici les résultats produit par ce programme.

```
TN CN FN QN KN FN CN TN
PN PN PN PN PN PN PN PN
VD VD VD VD VD VD VD VD
VD VD VD VD VD VD VD VD
VD VD VD VD VD VD VD VD
VD VD VD VD VD VD VD VD
PB PB PB PB PB PB PB PB
TB CB FB QB KB FB CB TB

-----
TN CN FN QN KN FN CN TN
PN PN PN PN PN PN PN PN
VD VD VD VD VD VD VD VD
VD VD VD VD VD VD VD VD
VD VD VD VD VD VD VD VD
VD VD VD VD PB VD VD VD
PB PB PB PB VD PB PB PB
TB CB FB QB KB FB CB TB
```

ATTENTION : selon les personnes, il peut y avoir matière à débat quant à la terminologie de « tableaux multidimensionnels ». Effectivement, dans certains langages on peut écrire quelque chose de proche de cette déclaration : `int [,] multiDimArray = new Array[5,5];`. Dans ce cas, deux index sont nécessaires pour l'accès à une donnée. **Pour autant, cette syntaxe, bien que très sympathique, n'existe pas en Java.** Du coup, les personnes un peu strictes quant à la terminologie, préféreront parler de tableau de tableau. Je ne sais pas si vous l'avez remarqué, mais c'est la terminologie que j'ai plutôt privilégiée : et oui, je fais plutôt partie de cette catégorie :-). Pour autant, je note aussi que ce débat ne pose aucun souci éthique à société mère de Java qui parle sans aucun problème de tableaux multidimensionnels, comme en atteste ce lien. Du coup, je vous recommande aussi ce document (et surtout la partie sur le clonage de tableaux) et je vous laisserais donc vous faire votre propre opinion.



Tableaux Java VS Collections Java

En Java, il n'y a pas que les tableaux qui permettent de stocker un ensemble de valeurs. Il y a aussi les collections Java. En fait, ces collections sont des classes qui, pour les principales, sont localisées dans le paquetage `java.util`. Ces classes sont dites génériques, car elles permettent de typer les éléments qui seront stockés dans ces collections. Nous reviendrons dans un futur chapitre sur cette notion de classes génériques.

Une des différences entre un tableau Java et une collection Java, réside dans le fait qu'un tableau est dimensionné une fois pour toute lors de sa déclaration (si vous dépassez des bornes, une exception sera déclenchée, comme nous l'avons déjà vu plus haut dans ce document) alors qu'une collection est dynamique (tant qu'il y a de la mémoire de disponible, on peut continuer à ajouter des éléments dans une collection).

Un autre point remarquable est qu'il existe différentes implémentations (différentes classes). En fonction de la classe choisie, vous aurez aussi choisi un algorithme, associé à cette classe, ce qui pourra avoir des impacts sur les performances si vous n'avez pas choisi le bon. Par exemple, la classe `java.util.ArrayList` implémente un tableau d'éléments, mais extensible en fonction des besoins. Au contraire, la classe `java.util.LinkedList` implémente un algorithme de liste doublement chaînée.

Voici un petit exemple d'utilisation d'une collection Java de type `java.util.ArrayList`. Notez bien que lors de la déclaration de notre collection, aucune information de taille n'est spécifiée.

```
1 import java.util.ArrayList;
2
3 public class Demo {
4
5     public static void main( String [] args ) {
6
7         // Création de notre collection et ajouts de données
8         ArrayList<String> coll = new ArrayList<>();
9         coll.add( "azerty" );
10        coll.add( "qwerty" );
11
12        // Combien y a t'il de données dans la collection ?
13        System.out.println( "Size == " + coll.size() );
14
15        // Affichage des données de la collection
16        for (String string : coll) {
17            System.out.println( string );
18        }
19    }
20 }
21
22 }
```

Exemple d'utilisation d'une collection de type `java.util.ArrayList`

La ligne la plus importante est la 8 : c'est elle qui déclare et instancie notre collection. Notez-y la présence de typage `<String>`. Nous avons donc à faire à une collection de chaînes de caractères. Avant Java SE 7.0, il était nécessaire de rappeler ce typage au niveau du `new`. Aujourd'hui, un simple `<>` suffit, vu qu'il est déjà mentionné sur la déclaration de la variable.

La ligne 13 nous montre qu'il faut maintenant invoquer la méthode `size()` pour obtenir la taille de la collection. Ici, une taille de deux.

Enfin, remarquez en ligne 16 qu'on peut parcourir la collection avec la même syntaxe « for each » que pour un tableau classique. Voici les résultats produits par l'exemple précédent.

```
Size == 2
azerty
qwerty
```

Pour finir ce chapitre, voici un dernier programme un peu idiot. Une boucle infinie (`true` est toujours vrai) pousse des nouvelles données dans une collection Java. Comme celle-ci est extensible, elle ne cesse de grossir jusqu'à ... saturer la mémoire et produire une erreur grave.

```
1 import java.util.ArrayList;
2
3 public class Demo {
4
5     public static void main( String [] args ) {
6
7         long counter = 0;
8         ArrayList<String> collection = new ArrayList<>();
9         while ( true ) {
10            collection.add( "Value " + counter );
11            if ( counter % 1_000_000 == 0 ) {
12                System.out.printf( "counter == %d\n", counter );
13            }
14            counter++;
15        }
16    }
17 }
18
19 }
```

On test l'aspect extensible d'une collection Java

Et voici les dernières lignes produites sur la console, par le programme ci-dessus.

```
counter == 41 000 000
counter == 42 000 000
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
at Demo.main(Demo.java:10)
```