Un tableau est un ensemble indexé de données d'un même type. L'utilisation d'un tableau se décompose en trois parties :

- Création du tableau ;
- Remplissage du tableau ;
- Lecture du tableau.

Création d'un tableau

Un tableau se déclare et s'instancie comme une classe :

```
int monTableau[ ] = new int[10];
```

ou

```
int [ ] monTableau = new int[10];
```

L'opérateur [ ] permet d'indiquer qu'on est en train de déclarer un tableau.

Dans l'instruction précédente, nous déclarons un tableau d'entiers (int, integer) de taille 10, c'est-à-dire que nous pourrons stocker 10 entiers dans ce tableau.

Si [ ] suit le type, toutes les variables déclarées seront des tableaux, alors que si [ ] suit le nom de la variable, seule celle-ci est un tableau :

```
int [] premierTableau, deuxiemeTableau;
float troisiemeTableau[], variable;
```

Dans ces quatre déclarations, seule *variable* n'est pas un tableau.

Remplissage d'un tableau

Une fois le tableau déclaré et instancié, nous pouvons le remplir :

```
int [] monTableau = new int[10];

monTableau[5] = 23;
```

L'indexation démarre à partir de 0, ce qui veut dire que, pour un tableau de N éléments, la numérotation va de 0 à N-1.

Dans l'exemple ci-dessus, la 6ème case contient donc la valeur 23.

Nous pouvons également créer un tableau en énumérant son contenu :

```
int [] monTableau = {5,8,6,0,7};
```

Ce tableau contient 5 éléments.

Lorsque la variable est déjà déclarée, nous pouvons lui assigner d'autres valeurs en utilisant l'opérateur `new` :

```
monTableau = new int[]{11,13,17,19,23,29};
```

Lecture d'un tableau

Pour lire ou écrire les valeurs d'un tableau, il faut ajouter l'indice entre crochets (`[` et `]`) à la suite du nom du tableau :

```
int [] monTableau = {2,3,5,7,11,23,17};
int nb;

monTableau[5] = 23;  // ->  2 3 5 7 11 23 17
nb = monTableau[4];  // 11
```

L'indice 0 désigne le premier élément du tableau.

L'attribut `length` d'un tableau donne sa longueur (le nombre d'éléments). Donc pour un tableau nommé `monTableau` l'indice du dernier élément est `monTableau.length-1`.

Ceci est particulièrement utile lorsque nous voulons parcourir les éléments d'un tableau.

```
for (int i = 0; i < monTableau.length; i++)
{
    int élément = monTableau[i];
    // traitement
}
```

Parcours des tableaux

Java 5 fournit un moyen plus court de parcourir un tableau.

L'exemple suivant réalise le traitement sur monTableau (tableau d'entiers) :

```
int[] monTableau = {150, 200, 250};
for (int élément : monTableau)
{
   // traitement
}
```

Attention néanmoins, la variable `element` contient une copie de `monTableau[i]`. Avec des tableaux contenant des variables primitives, toute modification de `élément` n'aura aucun effet sur le contenu du tableau.

```
// Vaine tentative de remplir tous les éléments du tableau avec
la valeur 10
for(int élément : monTableau)
{
    élément = 10;
}

// La bonne méthode :
for(int i=0 ; i<monTableau.length ; i++)
{
    monTableau[i] = 10;
}

// Ou plus court :
Arrays.fill(monTableau, 10);
```

Pour éviter de modifier la variable, utilisez le mot-clé `final` :

```
for (final int élément : monTableau)
{
    // traitement
    // toute tentative de modification de la variable élément
sera détectée par le compilateur.
}
```

Tableaux à plusieurs dimensions

En Java, les tableaux à plusieurs dimensions sont en fait des tableaux de tableaux.

Exemple, pour allouer une matrice de 5 lignes de 6 colonnes :

```
int[][] matrice = new int[5][];
for (int i=0 ; i<matrice.length ; i++)
    matrice[i] = new int[6];
```

Java permet de résumer l'opération précédente en :

```
int[][] matrice=new int[5][6];
```

La première version montre qu'il est possible de créer un tableau de tableaux n'ayant pas forcément tous la même dimension.

On peut également remplir le tableau à la déclaration et laisser le compilateur déterminer les dimensions des tableaux, en imbriquant les accolades :

```
 int[][] matrice =
    {
        { 0, 1, 4, 3 } , // tableau [0] de int
        { 5, 7, 9, 11, 13, 15, 17 } // tableau [1] de int
    };
```

Pour déterminer la longueur des tableaux, on utilise également l'attribut `length` :

```
matrice.length     //  2
matrice[0].length  //  4
matrice[1].length  //  7
```

De la même manière que précédement, on peut facilement parcourir tous les éléments d'un tableau :

```
for (int i=0 ; i<matrice.length ; i++)
{
    for (int j=0 ; j<matrice[i].length ; j++)
    {
        //Action sur matrice[i][j]
    }
}
```

Depuis Java 5, il est possible de parcourir les valeurs comme ceci :

```
for (int[] row : matrice)
{
    for (int j=0 ; j<row.length ; j++)
    {
```

```
            //Action sur row[j]
        }
    }
```

Le parcours des éléments du tableau `row` peut également utiliser la boucle for itérative sur le type primitif `int`. Ce type de boucle ne permet pas de modifier les éléments du tableau.

```
for (int[] row : matrice)
{
    // Modifications sur row[index] répercutées sur matrice[...]
[index]
    // Modifications sur row ignorées (copie locale de la
référence au tableau)
    for (int cell : row)
    {
        // Action sur cell
        // Modifications sur cell ignorées (copie locale de la
valeur)
    }
}
```

Pour une matrice d'objet, cela est donc également possible :

```
String[][] matrice_de_themes =
{
    { "Java", "Swing", "JavaFX" },
    { "Python", "Numpy" },
    { "Vélo", "Chambre à air", "Rustine", "Guidon" },
    { "Cuisine", "Recette", "Ingrédient", "Préparation",
"Ustensile" },
};
for (String[] ligne_theme : matrice_de_themes)
{
    for (String mot : ligne_theme)
    {
        //Action sur mot
        System.out.println(mot);
    }
}
```

La classe Arrays

La classe `Arrays` du package `java.util` possède plusieurs méthodes de gestion des tableaux de types de base, et d'objets :

- la méthode `asList` convertit un tableau en liste,

- la méthode `binarySearch` effectue la recherche binaire d'une valeur dans un tableau,
- la méthode `equals` compare deux tableaux renvoie un booléen *true* s'ils ont même longueur et contenu,
- la méthode `fill` remplit un tableau avec la valeur donnée,
- la méthode `sort` trie un tableau dans l'ordre croissant de ses éléments.

```
String[] mots_clés =
{
    "Série",
    "Auteur",
    "Épisode",
    "Comédien",
    "Acteur",
    "Film",
    "Cinéma",
    "Réalisateur"
};
Arrays.sort(mots_clés);
for(final String mot :
mots_clés)
    System.out.println(mot);
```

Sortie console :

```
Acteur
Auteur
Cinéma
Comédien
Film
Réalisateur
Série
Épisode
```

Par défaut, `Arrays.sort()` place les caractères non ASCII après le "z" (ex : à, é...). Pour tenir compte de l'Unicode, il faut donc utiliser son deuxième paramètre : un `java.text.Collator` ou un `java.util.Comparator`.

```
String[] mots_clés =
{
    "Série",
    "Auteur",
    "Épisode",
    "Comédien",
    "Acteur",
    "Film",
    "Cinéma",
    "Réalisateur"
};

String règles_français =
    "
<a,A<b,B<c,C<d,D<e,E<f,F<g,G<h,H<i,I"
+
    "
<j,J<k,K<l,L<m,M<n,N<o,O<p,P<q,Q<r,R"
+
```

Sortie console :

```
Acteur
Auteur
Cinéma
Comédien
Épisode
Film
Réalisateur
Série
```

```
    "<s,S<t,T<u,U<v,V<w,W<x,X<y,Y<z,Z"
+
    "<\u00E6,\u00C6 aa,AA";
RuleBasedCollator collatorFrançais =
    new
RuleBasedCollator(règles_français);

Arrays.sort(mots_clés,
collatorFrançais);
for(final String mot : mots_clés)
    System.out.println(mot);
```

Copie d'un tableau

La copie d'un tableau implique la copie de ses éléments dans un autre tableau. Dans le cas d'un tableau d'objets, seules les références à ces objets sont copiées, aucun nouvel objet n'est créé.

La méthode `arraycopy` de la classe `System` permet de copier tout ou partie d'un tableau vers un autre tableau déjà alloué.

Comme toutes les classes, les tableaux dérivent de la classe `java.lang.Object`. Les méthodes de la classe `Object` sont donc utilisables :

```
int[] premiers = { 2, 3, 5, 7, 11 };
System.out.println( premiers.toString() ); // Par défaut
<type>@<hashcode>, exemple : [I@a108298c
System.out.println( Arrays.toString(premiers) );  // Pour
afficher son contenu à l'écran
```

La copie intégrale d'un tableau dans un nouveau tableau peut donc se faire en utilisant la méthode `clone()`. La valeur retournée par cette méthode étant de type `Object`, il faut la convertir dans le type concerné.

Exemple :

```
int[] nombres = { 2, 3, 5, 7, 11 };
int[] copie = (int[]) nombres.clone();
nombres[1]=4; // nombres contient 2 4 5 7 11
// tandis que copie contient toujours 2 3 5 7 11
```

Somme des éléments d'un tableau

```
public class SommeTableaux
{
```

```
    public static void main(String[] args)
    {
        int[] nb = {1,2,3,4,5,6,7,8,9};
        int somme = java.util.stream.IntStream.of(nb).sum();
        System.out.println(somme); //45
    }
}
```

Modifier la taille et copier une partie

La taille d'un tableau est fixe, mais il est possible d'allouer un autre tableau avec une taille différente et de recopier une partie des éléments du tableau d'origine. Cela peut se faire manuellement, ou en utilisant les méthodes existantes de la classe `java.util.Arrays`.

`Arrays.copyOf(T[] original, int nouvelle_taille)`
: Retourne un nouveau tableau de la taille spécifiée rempli avec les éléments du tableau original. Si la taille est plus petite, les derniers éléments ne sont pas copiés. Si la taille est plus grande, la fin du nouveau tableau est remplie par l'élément nul du type des éléments : `false` pour `boolean`, `0` pour les types numériques, `null` pour tous les autres types (tableau, chaîne, objet, ...).

Une autre méthode permet de copier une portion d'un tableau en spécifiant un index de début et un index de fin. Cet index de fin peut aller au delà de la taille du tableau original, auquel cas les derniers éléments du nouveau tableau sont remplis par l'élément nul du type des éléments, comme pour la méthode précédente.

`Arrays.copyOfRange(T[] original, int index_début, int index_fin)`
: Retourne un nouveau tableau dont la taille est la différence entre les deux indexs, rempli avec les éléments du tableau original dont l'indice est compris entre l'index de début (inclus) et celui de fin (exclu).

Comparaison de deux tableaux

Référence

Une variable de type tableau étant de type référence, l'utilisation des opérateurs de comparaison `==` et `!=` ne font que comparer les références. Ils ne permettent pas de détecter des références à deux tableaux différents mais dont la taille et le contenu est le même (éléments dans le même ordre).

```
import java.util.*;
public class CompareTableaux
{
    public static void main(String[] args)
    {
        String[]
            arr1 = { "2", "3", "5", "7", "11" },
            arr2 = { "2", "4", "6", "8", "11", "12" },
            arr3 = { "2", "4", "6", "8", "11", "12" }, // Même
contenu que arr2
```

```
            arr4 = arr3;                                    // Même
référence que arr3

        System.out.println("arr1 == arr2 --> " + (arr1 ==
arr2)); // false
        System.out.println("arr2 == arr3 --> " + (arr2 ==
arr3)); // false
        System.out.println("arr3 == arr4 --> " + (arr3 ==
arr4)); // true
    }
}
```

Contenu identique (éléments dans le même ordre)

Pour tester si deux tableaux ont la même taille et les mêmes éléments à la même position, le plus simple est d'utiliser la méthode statique `equals` de la classe `java.util.Arrays`. Cette méthode est en fait surchargée pour supporter les éléments de tous les types primitifs et des types références (objets).

```
import java.util.*;
public class CompareTableaux
{
    public static void main(String[] args)
    {
        String[]
            arr1 = { "2", "3", "5", "7", "11" },
            arr2 = { "2", "4", "6", "8", "11", "12" },
            arr3 = { "2", "4", "6", "8", "11", "12" }, // Même
contenu que arr2
            arr4 = arr3;                                    // Même
référence que arr3

        System.out.println("Arrays.equals(arr1, arr2) --> " +
Arrays.equals(arr1, arr2)); // false
        System.out.println("Arrays.equals(arr2, arr3) --> " +
Arrays.equals(arr2, arr3)); // true
        System.out.println("Arrays.equals(arr3, arr4) --> " +
Arrays.equals(arr3, arr4)); // true
    }
}
```

Éléments communs et particuliers

Il serait possible de lancer des boucles de comparaison, mais le plus court moyen donne un avant-goût du chapitre Collections. Le code ci-dessous établit, à partir de deux tableaux de chaînes de caractères, la liste des éléments communs aux deux et la liste des éléments particuliers à chaque tableau.

```java
import java.util.*;
public class CompareTableaux
{
    public static void main(String[] args)
    {
        String[]
            arr1 = { "2", "3", "5", "7", "11" },
            arr2 = { "2", "4", "6", "8", "11", "12" };

        List<String> l1 = Arrays.asList(arr1);
        List<String> l2 = Arrays.asList(arr2);

        // Intersection, comme un et logique :
        //   Éléments dans le tableau arr1 ET dans le tableau
arr2
        HashSet<String> communs = new HashSet<String>(l1);
        communs.retainAll(l2);

        // Union exclusive, comme un ou exclusif logique :
        //   Éléments dans le tableau arr1 OU dans le tableau
arr2 mais pas les deux.
        HashSet<String> non_communs = new HashSet<String>(l1);
        non_communs.addAll(l2);
        non_communs.removeAll(communs);

        System.out.println(communs);      // [11, 2]
        System.out.println(non_communs);  // [12, 3, 4, 5, 6, 7,
8]
    }
}
```

Nouvelle allocation d'un tableau de grande taille

Certaines applications ont besoin de beaucoup de mémoire et alloue beaucoup de ressources. Par exemple, un éditeur de vidéos peut garder en mémoire plusieurs images sous la forme d'un tableau de pixels pour modifier une séquence vidéo.

Dans une telle situation, la limite mémoire est souvent atteinte. Supposons que l'application ait alloué un tableau de pixels pour une opération qui vient de se terminer, et qu'un nouveau tableau de pixels de taille différente soit nécessaire pour la suite du traitement.

```java
public class VideoProcessing
{
    private int[] pixels;

    private void traitement()
    {
```

```
        // Traitement 1
        pixels = new int[width*height*2]; // HD, 16
bits/couleurs
        // ...

        // Traitement 2
        pixels = new int[width2*height2*2];
        // ...
    }
}
```

Au moment de la seconde allocation, il est nécessaire que la quantité de mémoire disponible soit suffisante pour la taille du nouveau tableau. Durant l'allocation du second tableau, le premier tableau n'est pas libéré : l'opération `new` est effectuée avant l'assignation qui écrase la référence au premier tableau.

Afin d'éviter un problème de quantité de mémoire disponible insuffisante, il est recommandé d'assigner `null` à la référence au premier tableau (et à toutes les autres références à ce même tableau) afin que le ramasse-miettes puisse libérer ce tableau si besoin.

```
public class VideoProcessing
{
    private int[] pixels;

    private void traitement()
    {
        // Traitement 1
        pixels = new int[width*height*2]; // HD, 16
bits/couleurs
        // ...

        // Traitement 2
        pixels = null; // Premier tableau libérable si besoin,
avant new
        pixels = new int[width2*height2*2];
        // ...
    }
}
```
