Haute École de Bruxelles
École Supérieure d'Informatique

Bachelor en Informatique
Laboratoires Java - 1ère année

TD Tableaux

Résumé

Voyons ici les tableaux, une structure qui peut contenir plusieurs exemplaires de données similaires.

1 Les tableaux

Un **tableau** est une suite d'éléments de même type portant tous le même nom mais se distinguant les uns des autres par un indice.

L'**indice** est un entier donnant la position d'un élément dans la suite. Cet indice varie entre la position du premier élément et la position du dernier élément, ces positions correspondant aux **bornes de l'indice**.

Notons qu'il n'y a pas de «trou » : tous les éléments existent entre le premier et le dernier indice.

La **taille d'un tableau** est le nombre (strictement positif) de ses éléments.

Attention ! la taille d'un tableau ne peut pas être modifiée pendant son utilisation.

Souvent on utilise un tableau plus grand que le nombre utile de ses éléments. Seule une partie du tableau est utilisée. On parle alors de **taille physique (la taille maximale du tableau)** et de **taille logique (le nombre d'éléments effectivement utilisés)**.

1.1 Notation en algo

Pour **déclarer un tableau**, on écrit :

```
nomTableau : tableau [borneMin à borneMax] de TypeÉlément
```

où `TypeÉlément` est le type des éléments que l'on trouvera dans le tableau. Les éléments sont d'un des types élémentaires vus précédemment (`entier`, `réel`, `booléen`, `chaine`, `caractère`) ou encore des variables structurées.

À ce propos, remarquons aussi qu'un tableau peut être un champ d'une structure. D'autres possibilités apparaitront lors de l'étude de l'orienté objet.

Les **bornes** apparaissant dans la déclaration sont des constantes ou des paramètres ayant une valeur connue lors de la déclaration. Une fois un tableau déclaré, **seuls les éléments d'indice compris entre borneMin et borneMax peuvent être utilisés**.

Par exemple, si on déclare :

```
tabEntiers : tableau [1 à 100] d'entiers
```

Il est interdit d'utiliser tabEntiers[0] ou tabEntiers[101]. De plus, chaque élément tabEntiers[i] (avec $1 \leq i \leq 100$) doit être manié avec la même précaution qu'une variable simple, c'est-à- dire qu'on ne peut utiliser un élément du tableau qui n'aurait pas été préalablement affecté ou initialisé.

N.B. : Il n'est pas interdit de prendre 0 pour la borne inférieure ou même d'utiliser des bornes négatives (par exemple : tabTempératures : tableau [-20 à 50] de réels). En Java, un tableau est défini par sa taille n et les bornes sont automatiquement 0 et $n - 1$. Ce n'est pas le cas en algorithmique où on a plus de liberté dans le choix des bornes

```
// Calcule et affiche la quantité vendue de 10 produits.
module statistiquesVentesAvecTableau()
    cpt : tableau [1 à 10] d'entiers
    i, numéroProduit, quantité : entiers

    pour i de 1 à 10 faire
      cpt[i] ← 0
    fin pour

    afficher "Introduisez le numéro du produit :"
    lire numéroProduit
    tant que numéroProduit > 0 faire
      afficher "Introduisez la quantité vendue :"
      lire quantité
      cpt[numéroProduit] ← cpt[numéroProduit] + quantité
      afficher "Introduisez le numéro du produit :"
      lire numéroProduit
    fin tant que

    pour i de 1 à 10 faire
      afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour
fin module
```

1.2 Déclaration et création en Java

Nous présentons ici une vue simplifiée des tableaux en Java afin de coller au cours d'algorithmique.

Nous aurons l'occasion d'être plus précis en DEV2.

Pour rappel, il est nécessaire de manipuler plusieurs variables similaires auxquelles on accède par un indice :

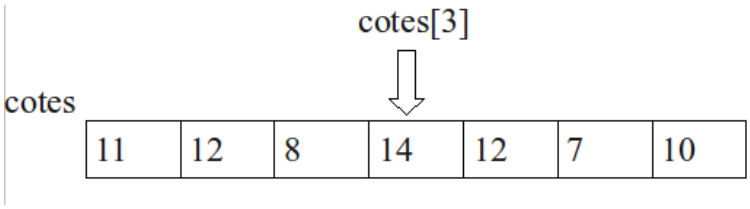

FIGURE 1 – tabPres.png

Contrairement à algo où on a le choix des valeurs desbornes pour les indices, en Java, les indices varient de `0` à `taille du tableau - 1`.

0 est l'indice de départ.

Déclaration et création : 2 étapes

En Java, l'étape de création est séparée de l'étape de déclaration.

En effet, l'étape de déclaration réserve un emplacement mémoire sur la pile qui contiendra une adresse où trouver la valeur des éléments du tableau. La création connaitra lataille du tableau, le créera sur le tas et remplira l'adresse sur la pile.

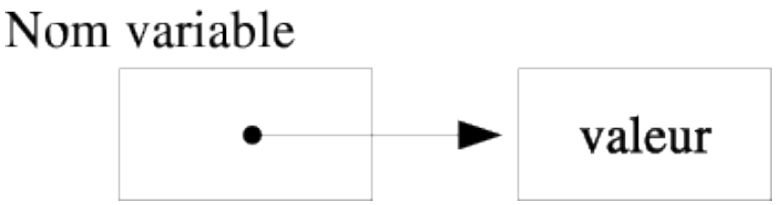

FIGURE 2 – reference.png

Déclaration

Pour déclarer un tableau :

```
Type[] identifier
```

Par exemple :

`int []` est le type tableau d'entiers

`String []` est le type tableau de chaines de caractères

```
int [] cotes ;
String [] noms;
```

Création

Pour créer un tableau :

```
identifier = new Type[taille]
```

Par exemple :

`new int[3]`

`new String[taille]` où `taille` est défini

```
int [] entiers ; // déclaration
entiers = new int[3]; // création
```

La déclaration et la création peuvent être combinées

```
int [] entiers = new int[3];
```

Initialisation

Par défaut, les éléments sont initialisés à 0 (numériques) ou false (booléens). Ce ne sont pas forcément les valeurs initiales que nous désirons. Pour changer ça :

```
identifier = new Type[] {x, x}
```

Par exemple :

`new int[] {42, 17, -5}`

`new String[] {"foo", "bar"}`

```
int [] entiers = new int[] {0x2A, 021, −5};
String [] noms = new String[] {" Victoria ", "Melanie", "Melanie", "Emma", "Geri"};
double[] réels ;
réels = new double[] {4.2, −1};
```

Cas particulier : déclaration, création et initialisation

On peut déclarer le tableau, le créer et l'initialiser en une seule étape, en donnant ses valeurs :

```
int [] entiers = {0x2A, 021, -5};
double[] pseudoRéels = {4.5, 1E-4, -4.12, Math.PI};

// Mais si sur 2 lignes :
double[] réels ;
réels = {4.2, -1}; // FAUX
```

Accès aux éléments

1. 0 est l'indice de départ
2. les indices varient de 0 à taille du tableau - 1
3. la taille du tableau est son nombre d'éléments

```
int [] entiers = {3, 14, 15};
int entier = entiers [2]; // entier vaut 15
entiers [1] = 85;
entier = 0;
entier = entiers [ entier +1]; // entier vaut 85
```

Par exemple :

```
package be.heb.esi. lg1 . tutorials . tableaux ;

public class InitialisationTableau {
    public static void main(String [] args) {
      int [] entiers = new int[10];
      for(int i = 0; i < 10; i++) {
        entiers [ i ] = i;
      }
    }
}
```

1.3 Taille logique et physique

Parfois, on connait la taille d'un tableau lorsqu'on écrit l'algorithme (par exemple s'il s'agit de retenir les ventes pour les douze mois de l'année) mais ce n'est pas toujours le cas (par exemple, connait-on le nombre de produits vendus par le magasin ?).

Si cette taille n'est pas connue, une possibilité est d'attribuer au tableau une taille maximale (sa taille physique) et de retenir dans une variable le nombre réel de cases utilisées (sa taille logique).

```
// Calcule et affiche la quantité vendue de x produits.
module statistiquesVentes()
    cpt : tableau [1 à 1000] d'entiers
    i, numéroProduit, quantité : entiers
    nbArticles : entier

    lire nbArticles
    pour i de 1 à nbArticles faire
      cpt[i] ← 0
    fin pour

    afficher "Introduisez le numéro du produit :"
    lire numéroProduit
    tant que numéroProduit > 0 ET numéroProduit <= nbArticles faire
      afficher "Introduisez la quantité vendue :"
      lire quantité
      cpt[numéroProduit] ← cpt[numéroProduit] + quantité
      afficher "Introduisez le numéro du produit :"
      lire numéroProduit
    fin tant que

    pour i de 1 à nbArticles faire
      afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour
fin module
```

1.4 Taille d'un tableau en Java

En Java, un tableau connait sa taille

```
identifier.length
```

```
int [] entiers = {4, 5, 6};
int taille = entiers . length ;
System.out. println ( taille ); // écrit 3
```

Par exemple :

```
package be.heb.esi. lg1 . tutorials . tableaux ;

public class SimpleParcoursAscendant {
    public static void main(String [] args){
      int [] entiers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
      for(int i = 0; i < entiers . length ; i = i + 1) {
        System.out. println ( entiers [ i ]);
      }
    }
}
```

ou encore

```
package be.heb.esi. lg1 . tutorials . tableaux ;

public class SimpleParcoursDescendant {
    public static void main(String [] args){
      int [] entiers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
      for(int i = entiers . length − 1; i >= 0; i = i−1) {
        System.out. println ( entiers [ i ]);
      }
    }
}
```

1.5 Tableau et paramètres de module

Un tableau peut être passé en paramètre à un module mais qu'en est-il de sa taille ? Il serait utile de pouvoir appeler le même module avec des tableaux de tailles différentes. Pour permettre cela, la taille du tableau reçu en paramètre est déclarée avec une variable (qui peut être considéré comme un paramètre entrant).

Par exemple :

```
module afficherTaille(tabEntier↓ : tableau [1 à n] d'entiers)
    afficher "J'ai reçu un tableau de ", n, " éléments".
fin module
```

Ce `n` va prendre la taille précise du tableau utilisé à chaque appel et peut être utilisé dans le corps du module. Bien sûr il s'agit là de la **taille physique** du tableau.

Si une partie seulement du tableau doit être traitée, il convient de **passer également la taille logique en paramètre**.

Par exemple :

```
module afficherTailles(tabEntiers↓ : tableau [1 à n] d'entiers, tailleLogique : enti
    afficher "J'ai reçu un tableau rempli de ", tailleLogique, " éléments "
    afficher "sur ", n, " éléments au total."
fin module
```

Un module peut retourner un tableau.

```
// Crée un tableau statique d'entiers de taille 10, l'initialise à 0 et le retourne
module créerTableau() → tableau [1 à 10] d'entiers
    tab : tableau [1 à 10] d'entiers
    i : entier
    pour i de 1 à 10 faire
      tab[i] ← 0
    fin pour
    retourner tab
fin module

module principalAppelTableau()
    entiers : tableau [1 à 10] d'entiers
    i : entier
    entiers ← créerTableau()
    pour i de 1 à 10 faire
      afficher entiers[i]
    fin pour
fin module
```

Attention, il n'est pas possible de lire ou d'afficher un tableau en une seule instruction ; il faut des instructions de lecture ou d'affichage individuelles pour chacun de ses éléments.

1.6 Tableau et paramètres de méthodes

Un tableau peut être un paramètre d'une méthode.

Par exemple :

```
public static void afficher ( int [] entiers ) {
    for(int i = 0; i <entiers . length ; i ++) {
      System.out. println ( entiers [ i ]);
    }
}
```

L'appel pourrait être

```
int [] cotes = {12, 8, 10, 14, 9};
afficher ( cotes );
```

Passage de paramètre par valeur

En Java, passage de paramètre par valeur. Pour un tableau, cela signifie que l'on ne peut pas modifier le tableau dans son ensemble mais que l'on pourra modifier ses éléments.

Par exemple :

```
public static void remplir ( int [] entiers , int val ) {
    for(int i = 0; i <entiers . length ; i ++) {
      entiers [ i ] = val;
    }
}
```

L'appel pourrait être

```
int [] cotes = new int[16]; // Ne pas oublier de le créer !
remplir ( cotes , 20 );
```

MAIS :

```
public static void méthodeFausse( double[] réels ) {
    double[] réelsDePassage = {4.2, -7, Math.PI};
    réels = réelsDePassage; // INUTILE
}
```

Quel que soit l'appel, le tableau que l'on passe en paramètre ne sera pas modifié.

Un tableau peut être une valeur de retour

Par exemple :

```
ppublic static int [] créer ( int taille , int val ) {
    int [] entiers = new int[ taille ];
    for(int i = 0; i < taille ; i ++) {
      entiers [ i ] = val;
    }
    return entiers ;
}
```

L'appel pourrait être

```
int [] cotes = créer(16, 20);
```

1.7 Parcours d'un tableau à une dimension.

Soit le tableau `tab` déclaré ainsi

```
tab : tableau [1 à n] de T  // où T est un type quelconque
```

Envisageons d'abord le parcours complet et voyons ensuite les parcours avec arrêt prématuré.

Parcours complet.

Pour parcourir complètement un tableau, on peut utiliser la boucle `pour` comme dans l'algorithme suivant où «traiter »va dépendre du problème concret posé : afficher, modifier, sommer, . . .

```
// Parcours complet d'un tableau via une boucle pour
// Les déclarations sont omises pour ne pas alourdir les algorithmes.
pour i de 1 à n faire
    traiter tab[i]
fin pour
```

Parcours avec sortie prématurée.

Parfois, on ne doit pas forcément parcourir le tableau jusqu'au bout mais on pourra s'arrêter prématurément si une certaine condition est remplie. Par exemple :

1. on cherche la présence d'un élément et on vient de le trouver ;
2. on vérifie qu'il n'y a pas de 0 et on vient d'en trouver un.

La première étape est de transformer le `pour` en `tant que` ce qui donne l'algorithme

```
// Parcours complet d'un tableau via une boucle tant-que
i ← 1
tant que i ≤ n faire
    traiter tab[i]
    i ← i+1
fin tant que
```

On peut à présent introduire le test d'arrêt. Une contrainte est qu'on voudra, à la fin de la boucle, savoir si oui ou non on s'est arrêté prématurément et, si c'est le cas, à quel indice.

Il existe essentiellement deux solutions, avec ou sans variable booléenne. En général, la solution [A] sera plus claire si le test est court.

Parcours avec sortie prématurée sans variable booléenne

```
// Parcours partiel d'un tableau sans variable booléenne
i ← 1
tant que i ≤ n ET test sur tab[i] dit que on continue faire
    i ← i+1
fin tant que

si i > n alors
    // on est arrivé au bout
sinon
    // arrêt prématuré à l'indice i.
fin si
```

Il faut être attentif à **ne pas inverser les deux parties du test**. Il faut absolument vérifier que l'indice est bon avant de tester la valeur à cet indice.

On pourrait inverser les deux branches du si-sinon en inversant le test mais attention à ne pas tester `tab[i]` car `i` n'est peut-être pas valide.

Dans certains cas, le si-sinon peut se simplifier en un simple return d'une condition.

Par exemple :

```
// Indique si un zéro est présent dans le tableau
module contientZéro(tab : tableau [1 à n] d'entiers) → booléen
    i : entier
    i ← 1
    tant que i ≤ n ET tab[i] ≠ 0 faire
      i ← i+1
    fin tant que
    retourner i ≤ n // Si le test est vrai c'est qu'on a trouvé un 0
fin module
```

Parcours avec sortie prématurée avec variable booléenne

```
// Parcours partiel d'un tableau avec variable booléenne
i ← 1
trouvé ← faux
tant que i ≤ n ET NON trouvé faire
    si test sur tab[i] dit que on a trouvé alors
      trouvé ← vrai
    sinon
      i ← i+1
    fin si
fin tant que
// tester le booléen pour savoir si arrêt prématuré.
```

Attention à bien choisir un nom de booléen adapté au problème et à l'initialiser à la bonne valeur. Par exemple, si la variable s'appelle «continue »

1. initialiser la variable à vrai ;
2. le test de la boucle est «. . .ET continue » ;
3. mettre la variable à faux pour sortir de la boucle.

1.8 Erreurs fréquentes et exceptions lancées en Java

1. `NullPointerException` : si vous essayez d'accéder à un élément d'un tableau qui n'a pas été créé (le tableau vaut null dans ce cas) ;
2. `ArrayIndexOutOfBoundsException` : si vous donnez un indice qui n'existe pas (ex : tab [10] quand il n'y a que 10 éléments dans le tableau).

2 Exercices

Maintenant, mettons tout ça en pratique.

2.1 Compréhension d'algorithme

Pour ces exercices, nous vous demandons de comprendre des algorithmes donnés.

Compréhension

Que vont-ils afficher ?

—
```
module boucle1 ()
    x : entier
    x ← 0
    tant que x < 12 faire
      x ← x+2
    fin tant que
    afficher x
fin module
```

———

—
```
module boucle2 ()
    ok : booléen
    x : entier
    ok ← vrai
    x ← 5
    tant que ok faire
      x ← x+7
      ok ← x MOD 11 ≠ 0
    fin tant que
    afficher x
fin module
```

———

```
— module boucle3 ()
      ok : booléen
      cpt, x : entiers
      x ← 10
      cpt ← 0
      ok ← vrai
      tant que ok ET cpt < 3 faire
        si x MOD 2 = 0 alors
          x ← x+1
          ok ← x < 20
        sinon
          x ← x+3
          cpt ← cpt + 1
        fin si
      fin tant que
      afficher x
  fin module

  ———
— module boucle4 ()
      pair, grand : booléens
      p, x : entiers
      x ← 1
      p ← 1
      faire
        p ← 2*p
        x ← x+p
        pair ← x MOD 2 = 0
        grand ← x > 15
      jusqu'à ce que pair OU grand
      afficher x
  fin module

  ———
— module boucle5 ()
      i, x : entiers
      ok : booléen
      x ← 3
      ok ← vrai
      pour i de 1 à 5 faire
        x ← x+i
        ok ← ok ET (x MOD 2 = 0)
      fin pour
      si ok alors
        afficher x
      sinon
```

```
      afficher 2 * x
    fin si
fin module
```

```
module boucle6 ()
    i, j, fin : entiers
    pour i de 1 à 3 faire
      fin ← 6 * i - 11
      pour j de 1 à fin par 3 faire
        afficher 10 * i + j
      fin pour
    fin pour
fin module
```

2.2 Compréhension de codes Java

Instructions répétitives

Quelles instructions répétitives sont correctes parmi les suivantes ? Expliquez pourquoi les autres ne le sont pas.

- ❏ proposition 1

```
While ( condition ) {
    // instructions
}
```

- ❏ proposition 2

```
do while ( condition ) {
    // instructions
}
```

- ❏ proposition 3

```
while ( true ) {
    // instructions
}
```

- ❏ proposition 4

```
while ( true ) do {
    // instructions
}
```

- ❏ proposition 5

```
FOR ( int i=0; i<=10; i=i+2 ) DO {
    // instructions
}
```

❏ proposition 6

```
for ( int i=0; i<=10; i=i+2 ) {
    // instructions
}
```

❏ proposition 7

```
for ( int i=0; i<=10; i=i+2 ) do {
    // instructions
}
```

❏ proposition 8

```
for ( int i=9; i>=0; i=i−2 ) {
    // instructions
}
```

Activité 'remplir les blancs'

Quel opérateur de comparaison Java représente la relation suivante ?

1. "est égal à" ? ____
2. "est différent de" ? ____

Quel opérateur booléen Java représente l'opérateur logique suivant ?

1. le ET : ____
2. le OU : ____
3. le NON : __

Expérience

Indiquez l'affichage obtenu par ce code.

Compréhension

Que vont-ils afficher ?

```
public class Boucles {

    public static void main ( String[] args ) {
        int facteur;
        final int VALEUR = 3;

        for (facteur = 1 ; facteur <= 10 ; facteur++){
            System.out.print(facteur*VALEUR+" ");
        }
        System.out.println();
    }
}
```

Exercice Tant que

Écrivez en Java l'algorithme suivant.

```
MODULE Test

    nb, produit : Entier
    produit ← 1

    LIRE nb
    TANT QUE nb ≠ 0 FAIRE
        produit ← produit * nb
        LIRE nb
    FIN TANT QUE
    AFFICHER produit

FIN MODULE
```

Exercice Pour

Écrivez en Java l'algorithme suivant.

```
MODULE Test

    nb: Entier
    i : Entier

    LIRE nb
    POUR i DE 1 A nb FAIRE
        AFFICHER i
    FIN POUR

FIN MODULE
```

2.3 À vous de jouer...

Il est temps de se lancer et d'écrire vos premiers modules et programmes Java correspondant. Voici quelques conseils pour vous guider dans la résolution de tels problèmes :

— il convient d'abord de bien comprendre le problème posé ; assurez-vous qu'il est parfaitement spécifié ;

— déclarez ensuite les variables (et leur type) qui interviennent dans l'algorithme ; les noms des variables risquant de ne pas être suffisamment explicites ;
— **mettez en évidence les variables «données », les variables «résultats »et les variables de travail ;**
— n'hésitez pas à faire une ébauche de résolution en français avant d'élaborer l'algorithme définitif pseudo-codé.
— Écrivez la partie algorithmique **AVANT** de vous lancer dans la programmation en Java.

Écrivez les algorithmes et codez les programmes Java correspondant qui

1. reçoit un naturel `n` et affiche
   (a) les `n` premiers entiers strictement positifs ;
   (b) les `n` premiers entiers strictement positifs en ordre décroissant ;
   (c) les `n` premiers carrés parfaits ;
   (d) les `n` premiers naturels impairs ;
   (e) les naturels impairs qui sont inférieurs ou égaux à `n`.

   Si le `n` reçu n'est pas strictement positif, votre programme s'arrêtera en générant une erreur/une exception.
2. demande à l'utilisateur d'introduire un entier positif (non strictement). Ce module permet à l'utilisateur de se tromper à plusieurs reprises mais l'utilisateur devra donner une bonne valeur pour arrêter le programme. (On suppose tout de même que l'utilisateur ne donne que des valeurs entières !)
3. lit une série de nombres entiers positifs, jusqu'à ce que l'utilisateur encode la valeur 0. Les nombres multiples de 3 seront affichés au fur et à mesure et le nombre de ces multiples sera affiché en fin de traitement.

   Pensez à utiliser le module écrit ci-dessus qui permet de lire un entier positif.
4. retourne la somme des chiffres qui forment un nombre naturel `n` Attention, on donne au départ le nombre et pas ses chiffres. Exemple : 133045 doit donner comme résultat 16, car 1 + 3 + 3 + 0 + 4 + 5 = 16.
5. lit une suite de nombres positifs entrés au clavier et affiche le maximum, le minimum, leur somme et la moyenne.

   La fin de la suite de nombre sera signifiée par une valeur sentinelle que vous choisirez judicieusement.
6. vérifie si un entier donné forme un palindrome ou non. Un nombre palindrome est un nombre qui lu dans un sens (de gauche à droite)

est identique au nombre lu dans l'autre sens (de droite à gauche). Par exemple, 1047401 est un nombre palindrome.

7. affiche les `n` premiers termes de la suite suivante : `1, -1, 2, -3, 5, -8, 13, -21, 34, -55,` où `n` est un entier strictement positif reçu en paramètre.
   Si le `n` reçu n'est pas strictement positif, votre programme s'arrêtera en générant une erreur/une exception.

En java, n'oubliez pas d'écrire la javadoc et la méthode main pour tester vos méthodes.

Jeu de la fourchette

Écrivez un algorithme qui simule le jeu de la fourchette. Ce jeu consiste à essayer de découvrir un nombre quelconque compris entre 0 et 100 inclus, tiré au sort par l'ordinateur (la primitive `hasard(n : entier)` retourne un entier entre 0 inclus et n exclus).

L'utilisateur a droit à huit essais maximum.

À chaque essai, l'algorithme devra afficher un message indicatif

— «nombre donné trop petit »
— ou «nombre donné trop grand ».

En conclusion,

— soit «bravo, vous avez trouvé en [nombre] essai(s) »
— soit «désolé, le nombre était [valeur] ».

Écrivez le code java correspondant.

Aide en Java : un petit tour dans l'API de la classe Random devrait vous aider à trouver l'équivalent du `hasard(n : entier)` en Java