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.

version 2014-2015 (24 janvier 2016)

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 de taille 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.

Une fois un tableau déclaré, **seuls les éléments d'indice compris entre 0 et taille-1 peuvent être utilisés**.

Par exemple, si on déclare :

```
tabEntiers : tableau de 100 entiers
```

Il est interdit d'utiliser tabEntiers[-1] ou tabEntiers[100]. De plus, chaque élément tabEntiers[i] (avec $0 \leq i \leq 99$) 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 autre chose que 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.
algorithme statistiquesVentesAvecTableau()
    cpt : tableau de 10 entiers
    i, numéroProduit, quantité : entiers

    pour i de 0 à 9 faire
      cpt[i] ← 0
    fin pour

    afficher "Introduisez le numéro du produit :"
    demander numéroProduit
    tant que numéroProduit ≥ 0 et numéroProduit ≤ 9 faire
      afficher "Introduisez la quantité vendue :"
      demander quantité
      cpt[numéroProduit] ← cpt[numéroProduit] + quantité
      afficher "Introduisez le numéro du produit :"
      demander numéroProduit
    fin tant que

    pour i de 0 à 9 faire
      afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour
fin algorithme
```

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 :

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

0 est l'indice de départ.

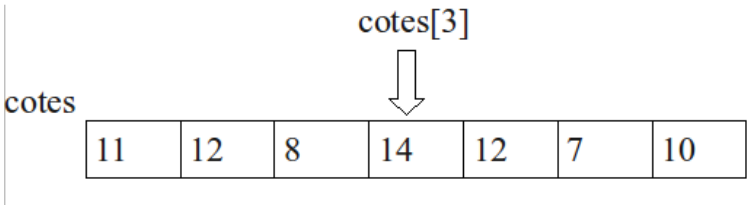

FIGURE 1 – tabPres.png

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 la taille du tableau, le créera sur le tas et remplira l'adresse sur la pile.

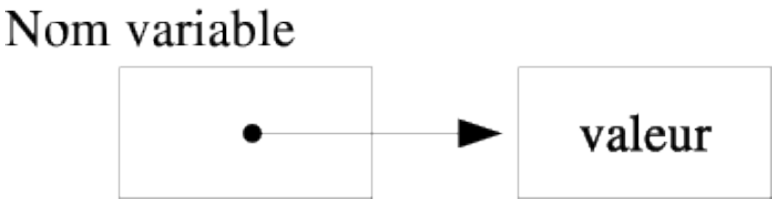

FIGURE 2 – reference.png

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.
algorithme statistiquesVentes()
    cpt : tableau de 1000 entiers
    i, numéroProduit, quantité : entiers
    nbArticles : entier

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

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

    pour i de 0 à nbArticles-1 faire
      afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour
fin algorithme
```

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 d'algorithme

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

Par exemple :

```
algorithme afficherTaille(tabEntier↓ : tableau de n entiers)
    afficher "J'ai reçu un tableau de ", n, " éléments".
fin algorithme
```

Ce `n` va prendre la taille précise du tableau utilisé à chaque appel et peut être utilisé dans le corps de l'algorithme. 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 :

```
algorithme afficherTailles(tabEntiers↓ : tableau de n entiers, tailleLogique : entie
    afficher "J'ai reçu un tableau rempli de ", tailleLogique, " éléments "
    afficher "sur ", n, " éléments au total."
fin algorithme
```

Un algorithme peut retourner un tableau.

```
// Crée un tableau statique d'entiers de taille 10, l'initialise à 0 et le retourne
algorithme créerTableau() → tableau de 10 entiers
    tab : tableau de 10 entiers
    i : entier
    pour i de 0 à 9 faire
      tab[i] ← 0
    fin pour
    retourner tab
fin algorithme

algorithme principalAppelTableau()
    entiers : tableau de 10 entiers
    i : entier
    entiers ← créerTableau()
    pour i de 0 à 9 faire
      afficher entiers[i]
    fin pour
fin algorithme
```

Attention, il n'est pas possible de demander 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 de n T  // où n est la taille du tableau et 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 0 à n-1 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 ← 0
tant que i ≤ n-1 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 ← 0
tant que i ≤ n-1 ET test sur tab[i] dit qu'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
algorithme contientZéro(tab : tableau [1 à n] d'entiers) → booléen
    i : entier
    i ← 0
    tant que i ≤ n-1 ET tab[i] ≠ 0 faire
      i ← i+1
    fin tant que
    retourner i ≤ n-1 // Si le test est vrai c'est qu'on a trouvé un 0
fin algorithme
```

Parcours avec sortie prématurée avec variable booléenne

```
// Parcours partiel d'un tableau avec variable booléenne
i ← 0
trouvé ← faux
tant que i ≤ n-1 ET NON trouvé faire
    si test sur tab[i] dit qu'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 Les tests

2.1 Tests unitaires

Les tests unitaires

Tous les programmes réels contiennent des bugs (erreurs, défauts), parfois même beaucoup. C'est inacceptable :

- Inconfort pour l'utilisateur
- Perte de temps, d'argent, de données, de matériel
- Voire danger pour la vie humaine, par exemple : l'échec du vol d'Ariane 5 (fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5).

On peut trouver différents types d'erreurs :

- à la compilation
- à l'exécution, le programme s'arrête
- à l'exécution, le programme fournit une mauvaise réponse

On pourrait aussi parler d'autres défauts

- Trop lent
- Trop gourmand en mémoire

Pour produire un logiciel sans bug il faut

- suivre une méthodologie éprouvée pour produire une première version avec peu de bugs (cf. Analyse)
- tester, tester et . . . tester encore ! Pour détecter ceux qui restent
  - Besoin d'outils pour nous aider de façon à tester de la manière la plus facile/rapide possible
  - Il existe plusieurs sortes de tests : unitaires, d'intégration, fonctionnels, non-régression, . . .
  - Nous nous intéressons aux tests unitaires (fr.wikipedia.org/wiki/Test_(informatique)) : c'est une procédure permettant de tester le bon fonctionnement d'un algorithme (une méthode), ce qui n'est pas forcément suffisant.

Le plan de tests

Il faut planifier les tests, c'est-à-dire se demander quoi tester en choisissant les cas intéressants / judicieux. Pour ça, il faut se baser sur les erreurs fréquentes et cela viendra avec l'expérience.

Nous avons besoin d'un plan reprenant les tests à effectuer

- Quelles valeurs de paramètres sont pertinentes ?
- Quel est le résultat attendu ?

Ce plan de tests doit être préparé pendant que l'on code (ou même avant et éventuellement par une autre personne).

Il permet de s'assurer que l'on teste tous les cas pertinents.

On ne peut pas tester toutes les valeurs possibles

- Il faut choisir des valeurs représentatives
  - Cas général / particuliers
  - Valeurs limites
- Il faut imaginer les cas qui pourraient mettre en évidence un défaut de la méthode : on s'inspire des erreurs les plus fréquentes en programmation :
  - On commence/arrête trop tôt/tard une boucle
  - On initialise mal une variable
  - Dans un test, on se trompe entre $<$ et $\leq$
  - Dans un test, on se trompe entre ET et OU
  - ...

  C'est un savoir-faire $\Rightarrow$ Importance de l'expérience

Par exemple : `public static int max(int n1, int n2, int n3) ...` qui calcule la valeur maximale de trois nombres.

Que tester en plus du cas général ?

- Le maximum est la première/dernière valeur
- Présence de nombres négatifs

Par exemple : plan de tests de la méthode `max` :

#	nombres	résultat	ce qui est testé
1	1, 3, 0	3	cas général
2	1, −3, −4	1	maximum au début
3	1, 3, 11	11	maximum à la fin
4	−1, −3, −4	−1	que des négatifs

- Quand tester ?
  - Le plus souvent possible

    - Erreur plus facile à identifier/corriger
    - Idéalement après chaque méthode écrite
- Que tester ?
    - Tout
    - Le nouveau code peut mettre en évidence un problème dans le code ancien (régression)
- Comment tester ?
    - Pas à la main : intenable
    - Besoin d'un outil automatisé : **JUnit**

JUnit

JUnit : outil pour automatiser les tests unitaires :

- Le programmeur fournit les tests,
- JUnit exécute tous les tests et
- établit un rapport détaillant les problèmes

La classe de test contient une méthode de test par cas. Cette méthode

- est autonome (ne reçoit rien, ne retourne rien)
- contient des affirmations
- fait appel à la méthode à tester
- compare le résultat attendu et le résultat obtenu

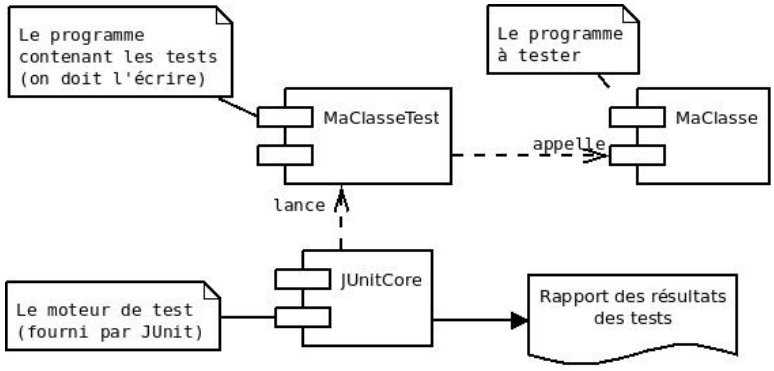

FIGURE 3 – junit.jpeg

Par exemple

```
@Test
public void max_cas1() {
    int n1 = 1, n2 = 3, n3 = 0;
    assertEquals ( 3, MaClasse.max(n1, n2, n3) );
}
```

— Reconnu comme un test unitaire grâce à l'annotation Test@
— Pas de `static`
— `assertEquals` vérifie que les 2 valeurs sont identiques
— `assertTrue ( val )`, `assertFalse ( val )`, . . .

Lancer les tests

Pour lancer le test, il suffit de compiler la classe de test puis de lancer la commande : `java org.junit.runner.JUnitCore package.MaClasseTest`.

Cette commande lance les tests et affiche un message récapitulatif du nombre de tests et du décompte de ceux qui ont réussi ou échoué.

Si cela ne fonctionne pas directement sur linux1 c'est que la JVM ne trouve pas la classe `org.junit.runner.JUnitCore` qui se trouve dans le fichier *jar*`/usr/share/java/junit4.jar`. Il faudra donc ajouter ce chemin à votre `CLASSPATH` et relancer un *shell* afin que la variable d'environnement soit relue.

Certains d'entre vous ne voudront pas entrer cette longue commande lors de chaque lancement de tests. Rien ne les empêche de définir un *alias* avec la commande :

```
alias javatest='java org.junit.runner.JUnitCore'
```

et ils n'auront plus alors qu'à exécuter la commande `javatest package.MaClasseTest` pour lancer la classe de tests.

3 Exercices

Maintenant, mettons tout ça en pratique.

3.1 En Java...

Bornes d'un tableau

— La variable permettant de se déplacer dans un tableau s'appelle un _____
— En Java, la première valeur d'un tableau à **n** éléments se trouve à la position __ et la dernière valeur se trouve à la position ____

Déclaration, création et initialisation d'un tableau

Un tableau possède un type et une taille. Le type de ses éléments est soit l'un des types simples (int, double, char,..) ou une réference (String,...).

La taille du tableau est le nombre (strictement positif) de ses éléments. Elle ne fait pas partie du type.

La déclaration d'un tableau réserve un emplacement mémoire

- ❍ sur le tas.
- ❍ sur la pile.

Création

Pour créer un tableau, le mot clé utilisé est ____

Pour pouvoir créer un tableau nommé **entiers** de 10 entiers **précédemment déclaré**, il faut écrire l'instruction _____ __ ____ ____ _____;

La création d'un tableau se fait

- ❍ sur le tas
- ❍ sur la pile

Initialisation

Le mot clé `new` permet également d'initialiser tous les éléments du tableau fraichement créé à une valeur par défaut.

Quelle est cette valeur par défaut ?

- — pour les numériques (int, double, char) : __
- — pour les boolean : _____
- — pour les objets (type réference) : _____

Réflexion

Est-il possible de déclarer, de créer et d'initialiser un tableau de 10 éléments en une seule opération ?

(la réponse est disponible dans la version en ligne)

Vérification

Écrivez sur papier la boucle permettant d'initialiser les éléments d'un tableau d'entiers de 10 éléments à la valeur de leur indice.

(la réponse est disponible dans la version en ligne)

Cochez les affirmations qui sont vraies

Considerons le tableau `entiers` contenant les élements `{3,4,5}`.

- ❏ Le tableau connait sa taille.
- ❏ `entiers[3]` vaut 5.
- ❏ `entiers[3]=4;` affecte la valeur 4 à la 3ème case.
- ❏ `entiers[1]=0;` supprime la case d'indice 1.
- ❏ Si `entiers20` est un tableau de 20 entiers, on peut écrire l'instruction `entiers20 = entiers;`.
- ❏ Un tableau peut être le seul paramètre d'une méthode.
- ❏ Un tableau peut être une valeur de retour d'une méthode.

Sélection multiple

Partons de la représentation de `entiers` en mémoire donnée par la figure ci-dessous.

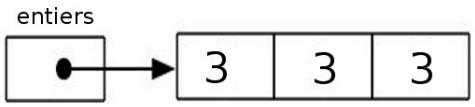

FIGURE 4 – représentation mémoire de entiers

Quelle(s) instruction(s) mène(nt) à ce résultat.

- ❏ proposition 1

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

- ❏ proposition 2

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

- ❏ proposition 3

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

- ❏ proposition 4

```
int [] entiers = {3,3,3};
```

- ❏ proposition 5

```
int [] entiers = new int [3];
for (int i=0; i<entiers.length; i++) {
    entiers[i] = 3;
}
```

❑ proposition 6

```
int [] entiers ;
entiers = {3,3,3};
```

❑ proposition 7

```
int [3] entiers = {3,3,3};
```

❑ proposition 8

```
int [] entiers ;
entiers = new {3,3,3};
```

Quiz

Pour ce tableau de 3 éléments, l'instruction `System.out.println (entiers[6]);`

❍ n'affiche rien.
❍ affiche une valeur se trouvant en mémoire.
❍ provoque une erreur à la compilation.
❍ provoque une erreur à l'exécution.

3.2 Les tests unitaires

Les tests

Le type de tests qui teste séparément chaque méthode est appelé tests _____ .

Avant de commencer à tester, il faut écrire un _____ de tests afin de s'assurer qu'on teste tous les cas.

Bien sûr, il n'est pas pensable de tester toutes les valeurs possibles, il faut donc choisir des valeurs représentatives (c-à-d cas général, cas particuliers et valeurs limites).

_____ est un outil pour automatiser les tests unitaires.

Cette méthode est autonome (ne reçoit rien ne retourne rien) et elle ne contient que des affirmations (appel de la méthode à tester, comparaison entre le résultat attendu et le résultat obtenu).

Une classe de test contient une méthode de test par cas. Chaque méthode de test est reconnue comme étant un test unitaire grâce au tag _____ . Elle contient des affirmations :

— _____ (intAttendu, intObtenu) vérifie que les 2 valeurs entières `intAttendu` et `intObtenus` sont identiques ;
— _____ (val) vérifie que la valeur du booléen `val` est à vrai,

— _____ (val) vérifie que la valeur de `val` est à faux.

Le tag _____ (_____) permet de vérifier que la méthode lance bien une exception de type `IllegalArgumentException`.

Dans la classe de tests, 2 instructions `import` sont nécessaires :

— import ____ . _____ . _____; // Pour que @Test soit connu
— import _____ ____ . _____ . _____ . __; // Pour que les méthodes de vérification ci-dessus soient reconnues.

La commande

---

permet de lancer les tests de la classe `ExerciceTest`, dont le package est `g32000.tds.tdTableaux`, précédemment compilée.

3.3 À vous de jouer...

N'oubliez pas nos 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 ;
— résolvez le problème via quelques exemples précis ;
— 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 ;
— déclarez ensuite les variables (et leur type) qui interviennent dans chaque algorithme ; les noms des variables risquant de ne pas être suffisamment explicites.
— Écrivez la partie algorithmique **AVANT** de vous lancer dans la programmation en Java.
— Demandez-vous si vous avez besoin de parcourir tout le tableau ou de sortir prématurément (si on a trouvé ce qu'on cherche par exemple).
— Pour la partie Java, dessinez l'arborescence des fichiers.
— **Écrivez le plan de tests en écrivant l'algorithme. Codez les tests après avoir écrit le code Java.**

Écrivez les algorithmes et codez les programmes Java correspondant qui

1. reçoit en paramètre le tableau `températures` de `n` réels et qui retourne le plus grand écart absolu entre deux températures consécutives de ce tableau. Et si on veut le plus petit écart ?
2. reçoit en paramètre le tableau valeurs de `n` entiers et qui vérifie si

ce tableau est ordonné (strictement) croissant sur les valeurs. L'algorithme retournera vrai si le tableau est ordonné, faux sinon.

3. reçoit en paramètre le tableau `tabCar` de `n` caractères, et qui «renverse »ce tableau, c'est-à-dire qui permute le premier élément avec le dernier, le deuxième élément avec l'avant-dernier et ainsi de suite.
4. reçoit en paramètre le tableau `tabChaines` de `n` chaines et qui vérifie si ce tableau est symétrique, c'est-à-dire si le premier élément est identique au dernier, le deuxième à l'avant-dernier et ainsi de suite.
5. reçoit un nombre entier positif ou nul en paramètre et qui affiche pour chacun de ses chiffres le nombre de fois qu'il apparait dans ce nombre. Ainsi, pour le nombre 10502851125, l'affichage mentionnera que le chiffre 0 apparait 2 fois, 1 apparait 3 fois, 2 apparait 2 fois, 5 apparait 3 fois et 8 apparait une fois (l'affichage ne mentionnera donc pas les chiffres qui n'apparaissent pas).
6. vérifie si un tableau d' entiers 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.

En java, n'oubliez pas d'écrire la javadoc et les tests de vos méthodes.

Mastermind

Dans le jeu du Mastermind, un joueur A doit trouver une combinaison de `k` pions de couleur, choisie et tenue secrète par un autre joueur B. Cette combinaison peut contenir éventuellement des pions de même couleur. À chaque proposition du joueur A, le joueur B indique le nombre de pions de la proposition qui sont corrects et bien placés et le nombre de pions corrects mais mal placés.

Pour implémenter une simulation de ce jeu, on utilise le type Chaine, dont les valeurs possibles sont les couleurs des pions utilisés. (Attention, le nombre exact de couleurs n'est pas précisé.) Les seules manipulations permises ont la comparaison (tester si deux couleurs sont identiques ou non) et l'affectation (affecter le contenu d'une variable de type Couleur à une autre variable de ce type). Les propositions du joueur A, ainsi que la combinaison secrète du joueur B sont contenues dans des tableaux de k composantes de type Chaine.

Écrire l'algorithme suivant qui renvoie dans les variables `bienPlacés` et `malPlacés` respectivement le nombre de pions bien placés et mal placés dans la «proposition »du joueur A en la comparant à la «solution »cachée du joueur B.

```
algorithme testerProposition( proposition↓, solution↓ : tableau de k chaines, bienPl
```

(une solution est disponible dans la version en ligne)

En java, n'oubliez pas d'écrire la javadoc et les tests de vos méthodes.