Université de Lille
IUT de Lille

TD n°9

R1.01 - Initiation au développement
BUT Informatique
Yann.Secq@univ-lille.fr
Fabien.Delecroix@univ-lille.fr
2022–2023

Objectifs: Comprendre les tableaux multi-dimensions, savoir les parcourir, les remplir, etc.

Rappel sur les tableaux multi-dimensionnels

Un tableau à plusieurs dimensions se déclare en `iJava` comme une variable quelconque en ajoutant autant de crochets, [], qu'il y a de dimensions. Si il n'y a qu'une paire de crochets, on obtient un tableau à une seule dimension, comme ce que l'on a utilisé dans le TD précédent. Si l'on a deux paires de crochets, on dispose alors d'un tableau à 2 dimensions que l'on peut considérer comme une grille dont la première dimension représente les lignes et la seconde les colonnes. Si l'on a 3 paires de crochets, c'est un tableau à 3 dimensions, c'est-à-dire nécessitant l'usage de 3 indices pour identifier une information. On peut visualiser ce tableau par un espace 3D dans lequel une case est représentée par des coordonnées (x, y, z). Si l'on passe à 4 paires de crochets, on peut associer la 4ième dimension au temps !

Pour l'instant, nous nous concentrerons sur les tableaux à 2 dimensions qui sont les plus souvent utilisés. Sachez cependant que vous pouvez définir des tableaux avec autant de dimensions que vous le souhaitez.

Nous allons étudier un exemple portant sur l'utilisation d'un tableau à deux dimensions de caractères.

Un tableau à deux dimensions peut être vu comme une grille dans laquelle on peut repérer une case avec un premier nombre, l, qui représente l'indice d'une ligne et un second nombre, c, qui représente l'indice d'une colonne, soit une coordonnée (l, c).

La déclaration d'une telle grille peut se faire ainsi: `char[][] grille;`

Une fois la déclaration faite, la phase d'allocation est similaire à un tableau à une dimension, si ce n'est qu'il est nécessaire de préciser le nombre de cases dont on souhaite disposer pour chaque dimension, c'est-à-dire le nombre de lignes et de colonnes dans notre exemple. Si l'on souhaite une grille rectangulaire de 2 lignes sur 4 colonnes: `grille = new char[2][4];`

Si l'on souhaite accéder à la case contenant le caractère c, on utilisera comme indice pour la première dimension, $ligne = 0$, et comme indice pour la seconde dimension, $colonne = 0$. Par exemple: `print(grille[0][0]);`

Pour obtenir le nombre de cases que contient un tableau à plusieurs dimensions, on ne peut plus utiliser simplement l'instruction `length(<type tableau> t)` → `int`.

En effet, il est maintenant nécessaire de préciser la dimension dont on souhaite connaître le nombre de cases. Ainsi, pour les tableaux multi-dimensionnels, on utilisera l'instruction:

`length(<type tableau> t, int dimension)` → `int`

dont le deuxième paramètre permet de préciser la dimension nous intéressant (de 1 à n). Avec le tableau définit précédemment, si l'on appelle `length(grille,1)`, cela revient à déterminer le nombre de lignes et retournera donc 2. Par contre, si l'on appelle `length(grille,2)`, cela revient à déterminer le nombre de colonnes et cela retournera donc 4.

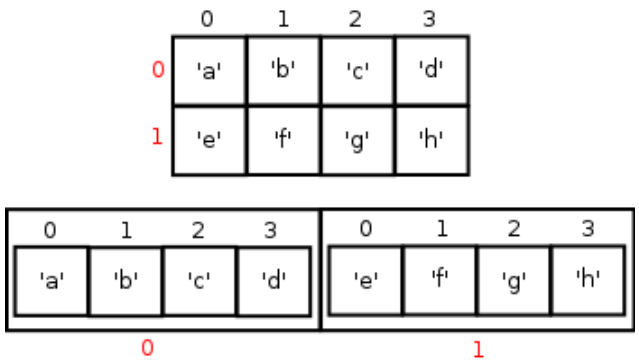

FIGURE 1 – Représentation intuitive et physique d'un tableau à deux dimensions

Pour vous aider à vous représenter un tableau à deux dimensions, vous trouverez ci-dessus deux représentations possibles : la première qui correspond à l'intuition, la seconde à ce qui se passe dans la mémoire de l'ordinateur. Pour nous, il est plus simple de considérer un tableau à deux dimensions comme une grille avec des lignes et des colonnes.

Par contre, pour la machine, la mémoire n'est qu'une suite de cases pouvant contenir 16, 32, 64 bits ... Un tableau à deux dimensions est donc juste un tableau à une dimension dont chaque case contient un tableau à une dimension ! Ainsi, l'on comprend peut-être mieux pourquoi le premier indice d'un tableau à deux dimensions correspond à celui des lignes, alors que le second correspond à celui des colonnes.

Les tests pour les tableaux multi-dimensions sont un peu plus difficiles à écrire car on ne dispose pas de primitive qui permet de comparer directement deux tels tableaux. C'est pourquoi les tests se faire sur les lignes du tableau une à une, en utilisant la fonction `assertArrayEquals` pour les tableaux à une dimension.

Exercice 1 : Saisir un tableau à 2 dimensions

On désire écrire une fonction permettant de créer un tableau d'entiers à deux dimensions avec des entiers saisis auprès de l'utilisateur.

1. dans la première version de l'algorithme, il faudra remplir le tableau ligne par ligne. La fonction doit passer ce test, en supposant que l'utilisateur saisit les nombres 2, 4, 6, 1, 3, 5, 0, 0, 10 dans cet ordre.

```
1 void testSaisirTab2DParLignes () {
2   int[][] tab = saisirTab2DParLignes(3,3);
3   assertArrayEquals(new int[]{2,4,6}, tab[0]);
4   assertArrayEquals(new int[]{1,3,5}, tab[1]);
5   assertArrayEquals(new int[]{0,0,10}, tab[2]);
6 }
```

2. dans la seconde version, on remplira le tableau colonne par colonne. La fonction doit passer ce test, en supposant que l'utilisateur saisit les nombres 2, 4, 6, 1, 3, 5, 0, 0, 10 dans cet ordre.

```
1 void testSaisirTab2DParColonnes () {
2   int[][] tab = saisirTab2DParColonnes(3,3);
3   assertArrayEquals(new int[]{2,1,0}, tab[0]);
4   assertArrayEquals(new int[]{4,3,0}, tab[1]);
5   assertArrayEquals(new int[]{6,5,10}, tab[2]);
6 }
```

Exercice 2 : Rechercher une valeur dans un tableau à deux dimensions

On désire concevoir un algorithme permettant de rechercher une valeur dans un tableau de caractères à deux dimensions.

1. Une première version de l'algorithme va afficher les indices de toutes les occurrences du caractère. Écrire la fonction `void chercherOccurrences(char[][] tab, char valeur)` telle que l'exécution de l'algorithme ci-dessous

```
1 void algorithm() {
2    char[][] matrice = new char[][]{{'a','b','c'},{'d','a','b'},{'a','e','d'}};
3    chercherOccurrences(matrice, 'a');
4 }
```

va afficher

```
occurrence trouvée en 0,0
occurrence trouvée en 1,1
occurrence trouvée en 2,0
```

2. Une deuxième version de l'algorithme va chercher les indices de la première occurrence dans le sens de lecture. La fonction à écrire doit passer ce test

```
1 void testRechercherPremiereOccurrence () {
2    char[][] tab = new char[][]{{'a','b','c'},{'d','a','b'},{'a','e','d'}};
3    assertArrayEquals(new int[]{0,1}, rechercherPremiereOccurrence(tab, 'b'));
4    assertArrayEquals(new int[]{1,0}, rechercherPremiereOccurrence(tab, 'd'));
5    assertArrayEquals(new int[]{2,1}, rechercherPremiereOccurrence(tab, 'e'));
6    assertArrayEquals(new int[]{-1,-1}, rechercherPremiereOccurrence(tab, 'x'));
7 }
```

Exercice 3 : Somme de deux matrices

En mathématiques, les matrices sont des tableaux à deux dimensions sur lesquels on fait un certain nombre d'opérations. On désire concevoir un algorithme qui calcule la somme de deux matrices de même taille (mais pas forcément carrées). Vous veillerez à ce que votre algorithme vérifie les contraintes imposées sur les dimensions des matrices avant d'effectuer le calcul.

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,m} \end{pmatrix}$$

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \cdots & a_{1,m} + b_{1,m} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \cdots & a_{2,m} + b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} + b_{n,1} & a_{n,2} + b_{n,2} & \cdots & a_{n,m} + b_{n,m} \end{pmatrix}$$

FIGURE 2 – Principe de calcul de la somme de deux matrices

La fonction à écrire doit passer ce test :

```
1  void testSommeMatrices () {
2      int[][] m1 = new int[][]{{1,4,0},{5,6,0}};
3      int[][] m2 = new int[][]{{1,2,0},{0,0,1}};
4      int[][] sommeAttendue = new int[][]{{2,6,0},{5,6,1}};
5      int[][] somme = sommeMatrices(m1,m2);
6      assertArrayEquals(sommeAttendue[0], somme[0]);
7      assertArrayEquals(sommeAttendue[1], somme[1]);
8
9      // Somme de matrices de tailles différentes
10     m1 = new int[][]{{1,4,0},{5,6,0}};
11     m2 = new int[][]{{1,2},{0,1},{0,3}};
12     sommeAttendue = new int[0][0]; //une matrice de taille 0
13     somme = sommeMatrices(m1,m2);
14     assertArrayEquals(sommeAttendue, somme);
15 }
```

Exercice 4 : Matrice symétrique

On désire écrire une fonction qui vérifie qu'une matrice carrée est symétrique par rapport à la première diagonale. Remarque: une matrice 1x1 est forcément symétrique :) Pour une matrice 2x2, il suffit de comparer deux valeurs ...

Vous devez écrire la fonction qui passe ce test. Vous supposerez que la matrice donnée en entrée est carrée et il n'y a pas besoin de vérifier cette condition

```
1  void testEstSymetrique () {
2      assertTrue(estSymetrique(new int[][]{{2,4},{4,3}}));
3      assertTrue(estSymetrique(new int[][]{{1,4,0},{4,6,0},{0,0,1}}));
4      assertFalse(estSymetrique(new int[][]{{1,2,0},{0,0,1},{1,2,3}}));
5  }
```

Prolongements

Exercice 5 : Produit de matrices

Après les additions, nous nous intéressons maintenant au calcul du produit de matrice. Contrairement à la somme, il n'est pas nécessaire que les matrices soient de même taille. Cependant, certaines contraintes doivent être vérifiées, vous veillerez donc à ce que votre algorithme les respectent avant d'effectuer le calcul du produit. Voici la définition formelle du produit de matrice:

Mais comme vous êtes plutôt des informaticiens, voici un exemple de calcul du produit (figure ci-après) qui repose sur l'accumulation des multiplications terme à terme des éléments de la ligne et de la colonne correspondant à l'élément que l'on souhaite déterminer:

Définition formelle

Soient $\mathbf{A} = a_{i,j\in[[1,m]]\times[[1,n]]} \in \mathfrak{M}_{m,n}$ et $\mathbf{B} = b_{i,j\in[[1,n]]\times[[1,p]]} \in \mathfrak{M}_{n,p}$ deux matrices. On **définit** le produit de **A** par **B** comme la matrice $m \times p$ suivante :

$$\mathbf{A}\cdot\mathbf{B} = \left(\sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}\right)_{i,j\in[[1,m]]\times[[1,p]]} = \begin{pmatrix} \sum_{k=1}^{n} a_{1,k}b_{k,1} & \sum_{k=1}^{n} a_{1,k}b_{k,2} & \cdots & \sum_{k=1}^{n} a_{1,k}b_{k,p} \\ \sum_{k=1}^{n} a_{2,k}b_{k,1} & \sum_{k=1}^{n} a_{2,k}b_{k,2} & \cdots & \sum_{k=1}^{n} a_{2,k}b_{k,p} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{m,k}b_{k,1} & \sum_{k=1}^{n} a_{m,k}b_{k,2} & \cdots & \sum_{k=1}^{n} a_{m,k}b_{k,p} \end{pmatrix}$$

FIGURE 3 – Définition formelle du produit de deux matrices

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 1 \\ 1 & 2 & 3 \end{pmatrix} \quad \begin{pmatrix} 1 & 4 & 0 \\ 5 & 6 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 11 & 16 & 0 \\ 0 & 0 & 1 \\ 11 & 16 & 3 \end{pmatrix}$$

FIGURE 4 – Exemple de calcul de produit de deux matrices

1. en supposant que vous disposez de deux matrices $m1$ et $m2$ déjà initialisées, calculez dans une matrice $m3$ le produit de $m1$ par $m2$.