Université Paris Cité **Programmation Orientée Objet**
L2 **Année 2022-2023**

TD – Séance n°11

Généricité

Introduction

Le but de ce TD est d'écrire un ensemble de classes et interfaces pour la gestion d'un tableau bidimensionnel (tableau de tableaux) à accès restreint. Le contenu de ce tableau peut être lu uniquement à travers un « accumulateur ». Un accumulateur est intuitivement un objet capable de parcourir les cases du tableau le long d'une certaine direction (par exemple une ligne, une colonne, une diagonale, etc.), et d'accumuler (par exemple sommer, multiplier, etc.) les valeurs lues pendant le parcours.

Définir les classes et interfaces suivantes (chacune est décrite plus en détail plus loin) :

1. **Une interface générique `Accumulator`** qui intuitivement représente des objets qui peuvent parcourir un ensemble de valeurs, et qui accumulent les valeurs lues pendant leur parcours.
2. **Une interface générique `AccFunction`** qui décrit le type des fonctions d'accumulation (i.e. les fonctions utilisées par les objets de type `Accumulator` pour mettre à jour la valeur accumulée à la lecture d'une nouvelle valeur). En particulier une fonction de type `AccFunction` sera passée en paramètre à chaque instance d'une classe qui implémente `Accumulator`, pour en définir la fonction d'accumulation.
3. **Une exception `InvalidContentException`**, qui signale un état invalide du tableau.
4. **Une classe générique `Matrix`** qui représente un tableau bidimensionnel d'éléments de type `T`, avec **une classe interne `MatrixScanner`** qui décrit les accumulateurs qui peuvent parcourir les objets de classe `Matrix` le long d'une direction.
5. **Une classe générique `FunctionalMatrix`** qui étend `Matrix` et impose des contraintes sur le contenu du tableau.
6. **Une classe `Q`** dont les méthodes manipulent et testent des objets de classe `Matrix`.

1 Matrix et ses itérateurs

Exercice 1 Écrire l'interface `Accumulator`. Il s'agit d'une interface générique ; son type variable `S` représente le type de la valeur accumulée pendant le parcours. Elle doit contenir les méthodes suivantes :

- Une méthode `accumulate` censée accumuler l'élément courant et se déplacer sur le prochain élément du parcours. Cette méthode reçoit un argument de type `S`, représentant intuitivement une valeur avec laquelle l'élément courant peut être combiné avant d'être accumulé.
- Une méthode `read` censée retourner la valeur accumulée.
- Une methode `isOver` qui retourne un booléen censé indiquer si le parcours est terminé (c'est-à-dire s'il n'y a plus d'éléments à parcourir).

Voici un exemple d'utilisation de l'interface `Accumulator<Integer>`. En supposant que `a` est une variable de type `Accumulator` :

```
while (!a.isOver()) {
  a.accumulate(2);
}
Integer i = a.read();
```

Dans le fragment de code ci-dessus, `a` lit l'ensemble des valeurs auxquels il est associé, un par un, et combine chaque valeur avec l'entier 2 avant de l'accumuler ; la valeur accumulée après avoir lu tous les valeurs de l'ensemble, est affectée à `i`. La fonction qui spécifie comment combiner la valeur lue avec 2, ainsi que comment mettre à jour la valeur accumulée à chaque nouvelle lecture, a été préalablement passée en paramètre au constructeur de `a`.

Exercice 2 Écrire l'interface `AccFunction`. Elle est une interface fonctionnelle générique et dépend de deux types variables : S est le type de la valeur accumulée, et T est le type des valeurs lues (i.e. intuitivement T est le type des valeurs contenues dans le tableau). Remarquer que les deux types peuvent être différents. Par exemple on peut lire un ensemble de valeurs de type `Double` et accumuler un `Boolean`. La méthode `apply` de cette interface a comme premier argument la valeur courante accumulée `acc` (donc de type `S`), comme deuxième argument une nouvelle valeur `ext` de type `S`, et comme troisième argument la valeur lue, `donnee`, de type `T`. La méthode est censée renvoyer la nouvelle valeur accumulée, après la contribution obtenue par la combinaison de `ext` avec `donnee`.

À titre d'exemple supposer que les valeurs parcourues soient de type Integer, et que l'accumulateur considéré ne fait que sommer l'ensemble des valeurs parcourues. Pour créer un tel accumulateur on lui passerait une instance de l'interface `AccFunction` qui implémente apply avec une fonction à trois arguments, tous de type Integer, et qui renvoie la somme du premier et du troisième argument (ignorant le deuxième).

Encore à titre d'exemple supposer que les valeurs parcourues soient de type Double, et que l'accumulateur considéré calcule $[x_1] + [x_2]^2 + \cdots + [x_n]^n$, où $[x_1], \ldots [x_n]$ sont les parties entières des valeurs parcourues (dans l'ordre du parcours). À sa création, on passerait alors à l'accumulateur une instance de `AccFunction` où la fonction `apply` a : un premier argument *acc* de type Integer (la valeur accumulée), un deuxième argument i de type Integer (l'exposant), et un troisième argument x de type Double (la nouvelle valeur lue), et retourne $acc + [x]^i$ (où [x] est la partie entière de x).

Exercice 3 Écrire l'exception `InvalidContentException` de type "checked".

Exercice 4 Écrire la classe `Matrix`. Elle a pour donnée interne un tableau de tableaux d'éléments de type variable `T`. Tous les champs doivent être privés, pour garantir l'encapsulation de la classe.

la classe `Matrix` doit contenir au moins :

- Le tableau de tableaux.
- La classe interne privée MatrixScanner décrite plus loin. IMPORTANT : pour bien définir cette classe, remarquer que le type de la valeur accumulée ne coïncide pas forcément avec le type des éléments du tableau ; remarquer également que sur un même objet de classe Matrix il faut pouvoir créer des accumulateurs de types différents (par exemple un pour accumuler une valeur de type Integer, un pour accumuler une valeur de type Boolean, etc.)

— Un constructeur qui initialise le tableau avec un autre tableau du même type, passé en paramètre.
— Une méthode (pas publique) boolean checkContent qui vérifie le contenu d'un tableau de tableaux d'éléments de type `T`. Elle est à utiliser dans le constructeur avant d'initialiser le tableau interne. Si checkContent renvoie `false`, le constructeur lancera l'exception InvalidContentException. Étant donné que les éléments du tableau sont d'un type variable non borné, cette fonction ne fait rien dans la classe `Matrix`, cependant elle pourra être redéfinie dans les classes filles de `Matrix` qui veulent imposer une contrainte sur le contenu du tableau.
— Deux méthodes publiques `nRows` et `nCols` qui renvoient le nombre de lignes et de colonnes.
— Une méthode publique `rowScanner` qui reçoit :
  — un indice de ligne `i`,
  — une fonction d'accumulation,
  — une valeur initiale pour la valeur accumulée,

  et retourne un objet de classe MatrixScanner construit avec la valeur initiale et la fonction d'accumulation passées en paramètre, pour parcourir la ligne i du tableau.
— une méthode publique `colScanner` qui se comporte comme `rowScanner`, mais où `i` est un indice de colonne.

La classe `Matrix` pourra contenir d'autres méthodes d'utilité si nécessaire, mais elle devront être toutes privées.

Exercice 5 Écrire la classe `MatrixScanner` comme membre privé de la classe `Matrix`. Elle implémente l'interface Accumulator, et est générique avec type variable S, représentant le type de la valeur accumulée.

`MatrixScanner` maintient :
— un champ qui stocke la valeur accumulée ;
— les coordonnées de la case du tableau couramment lue ;
— un nombre de cases dont l'accumulateur est censé avancer à chaque déplacement, sur chacune des deux dimensions (par exemple la valeur de ces champs pourrait indiquer que chaque déplacement avance de 2 cases sur les lignes et 3 cases sur les colonnes) ;
— la fonction d'accumulation, de type AccFunction (veiller à bien choisir les types variables pour cette invocation du type AccFunction) ;

Les valeurs de tous ces champs (y compris la fonction d'accumulation) seront passées en argument au constructeur de `MatrixScanner`.

Les méthodes de l'interface Accumulator seront implémentées comme suit :
— La méthode `accumulate` d'un objet `MatrixScanner` lit la valeur sous les coordonnées courantes et l'accumule (dans la variable qui stocke la valeur accumulée), en utilisant la fonction d'accumulation. Ensuite la méthode déplace les coordonnées courantes du nombre approprié de cases sur chaque dimension.
— La méthode `isOver` retourne faux si les coordonnées courantes sont encore dans la limite des bornes du tableau.
— La méthode `read` renvoie la valeur accumulée courante.

2 S'il reste du temps...

Exercice 6 Écrire la classe `FunctionalMatrix` qui hérite de `Matrix`. Il s'agit d'une classe générique qui dépend d'un type variable `T`, représentant le type des éléments du tableau. Ici `T` doit être numérique (i.e. posséder au moins toutes les méthodes de la classe `Number`).

`FunctionalMatrix` redéfinit la méthode `checkContent` pour vérifier que dans chaque colonne du tableau il y ait *au plus* une valeur différente de 0. (`0` n'est pas du type `T`, donc il faut réfléchir comment on peut comparer avec l'élément zéro du type `T`.)

De plus `FunctionalMatrix` fournit une fonction publique supplémentaire `content` qui renvoie un tableau (unidimensionnel) d'éléments de type de `T`. Ce tableau représente de manière compacte le contenu du tableau bidimensionnel : à chaque position `i` du tableau unidimensionnel on retrouvera l'unique valeur non nulle de la colonne `i` du tableau bidimensionnel. (On se rappelle qu'on avait dit au début de cet exercice que chaque colonne du tableau contient *au plus* une valeur différente de 0. Aussi, attention aux champs de la classe mère qui ne sont pas accessibles par la classe `FunctionalMatrix`. Attention également à la façon de retourner le tableau unidimensionnel de type `T`.)

Exercice 7 Écrire la classe `Q`. Elle doit contenir au moins deux méthodes statiques `dotProduct` et `lowerBound`, décrites ci-dessous.

- `dotProduct` reçoit en paramètre un objet `m` de classe `Matrix` avec éléments de type `Double`, un entier `i` et un tableau de double `b`, et retourne le produit scalaire entre la ligne `i` de `m` et le tableau `b`. (Se rappeler que le produit scalaire entre deux tableaux de la même taille $[t_1, \ldots, t_n]$ et $[s_1, \ldots, s_n]$ est défini comme $t_1 * s_1 + \cdots + t_n * s_n$.)
- `lowerBound` reçoit en paramètre un objet `m` de classe `Matrix` avec éléments de type `Double`, un entier `j` et un `double b`, et retourne vrai si et seulement si tous les éléments de la colonne `j` de `m` sont supérieures ou égales à `b`.

La classe `Q` contient enfin cette méthode `main` qui teste les classes :

```
public static void main (String args[]) throws InvalidContentException {
  Double[][] t = {{2.0,3.0,4.0},{5.0,6.0,7.0}};
  double[] b = {1.0,2.0,2.0};
  Matrix<Double> m = new Matrix<Double> (t);
  System.out.println(dotProduct(m, 1, b));
  System.out.println(lowerBound(m, 1, 4));

  Double[][] t1 = {{1.0,0.0,4.0},{0.0,6.0,0.0}};
  FunctionalMatrix<Double> m1 = new FunctionalMatrix<Double> (t1);
  Double[] fun = new Double[3];
  m1.content(fun);
  System.out.println(Arrays.toString(fun));
}
```

On s'attend à l'affichage suivante :

```
31.0
false
[1.0, 6.0, 4.0]
```