Faculté des Sciences
Aix*Marseille Université

Année universitaire 2020/2021

Site : ☒ Luminy ☐ St-Charles ☐ St-Jérôme ☐ Cht-Gombert ☒ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☒ 1er semestre ☐ 2ème semestre ☐ Session 2 Durée de l'épreuve : 1h30
Examen de : L2 Nom du diplôme : Licence d'informatique
Code du module : SIN3U02 Libellé du module : Programmation 2
Calculatrices autorisées : NON Documents autorisés : NON

---

1 Vecteurs

Vector
− elements : ArrayList<double>
+ Vector(dimension : int) + Vector() + dimension() : int + addElement(element : double) + getElement(index : int) : double + sum(vector : Vector) : Vector + product(scalar : double) : Vector

La classe `Vector` représente des vecteurs de réels de dimension quelconque. Les éléments d'un vecteur de la classe `Vector` sont stockés dans un une liste de type `ArrayList`. Ainsi, la taille de cette liste nous indique la dimension du vecteur.
Etant donné le vecteur $\mathbf{v}$, on notera $\mathbf{v}[i]$ le i-ème élément qui compose $\mathbf{v}$.

1.1 Déclaration des propriétés

Écrire le code de la classe `Vector` sans inclure les méthodes, c'est à dire en déclarant la classe et en indiquant ses propriétés.

1.2 Constructeurs

Écrire les constructeurs `public Vector(int dimension)` et `public Vector()`.
Le constructeur `Vector(int dimension)` crée un vecteur de avec `dimension` dimensions, dont tous les éléments sont à zéro. Le constructeur `Vector()` crée un vecteur vide et doit être rédigé en faisant appel au premier constructeur.

1.3 Ajout d'un élément

Écrire la fonction `public void addElement(double element)` qui ajoute l'élément `element` au vecteur, augmentant sa dimension de 1.

1.4 Accesseurs

Écrire les méthodes `public int dimension()` et `public double getElement(int index)`. La méthode `dimension` renvoie la dimension d'un vecteur, et la méthode `getElement` renvoie l'élément d'indice `index`. Si la valeur de l'indice est invalide, `getElement` doit lancer une exception de type `IndexOutOfBoundsException`.

1.5 Somme de deux vecteurs

Écrire la méthode `public Vector sum(Vector vector)` qui réalise la somme entre le vecteur courant et le vecteur en paramètre. Cette méthode retourne un nouveau vecteur. Si les deux vecteurs à additionner n'ont pas la même dimension, la méthode `sum` doit lancer une exception de type `InvalidParameterException`.
Pour $\mathbf{v}$, $\mathbf{v}'$ deux vecteurs de même dimension, la somme donnée par $\mathbf{v}.sum(\mathbf{v}')$ est un vecteur qui vérifie $\mathbf{v}.sum(\mathbf{v}')[i] = \mathbf{v}[i] + \mathbf{v}'[i]$, pour i tout indice valide pour $\mathbf{v}$ et $\mathbf{v}'$.

1.6 Produit d'un vecteur par un scalaire

Écrire la méthode `public Vector product(double scalar)` qui renvoie le produit du vecteur par le scalaire `scalar`. Cette méthode crée un nouveau vecteur. Pour $\mathbf{v}$ un vecteur et a un scalaire, le produit donné par $\mathbf{v}.product(a)$ est un vecteur qui vérifie $\mathbf{v}.product(a)[i] = \mathbf{v}[i] \times a$, pour i tout indice valide pour $\mathbf{v}$.

2 Matrices

Matrix
− elements : List<Vector>
+ Matrix(width : int, height : int) + Matrix() + width() : int + height() : int + addElement(vector : Vector) + getElement(index : int) : Vector + getElement(i : int, j : int) : double + sum(matrix : Matrix) : Matrix + product(scalar : double) : Matrix + product(vector : Vector) : Vector + product(matrix : Matrix) : Matrix

Une matrice peut être vue comme un tableau bimensionnel ou comme une collection de plusieurs vecteurs, chacun de ces vecteurs correspondant à une colonne du tableau. Nous les représenterons ici comme des collections de vecteurs. On dira qu'une matrice est de dimensions $m \times n$, c'est à dire de largeur m et de hauteur n, si elle est composée de m vecteurs de dimension n. Etant donné la matrice M, on notera $M[i]$ le i-ème vecteur qui compose M. Pour i un indicateur de numéro de colonne et j un indicateur de numéro de ligne, on note $M[i, j] = M[i][j]$ la valeur de la matrice M à la colonne i et la ligne j. Les vecteurs d'une matrice sont stockées dans une liste de type `ArrayList`.

2.1 Déclaration des propriétés

Écrire le code de la classe `Matrix` sans inclure les méthodes, c'est à dire en déclarant la classe et en indiquant ses propriétés.

2.2 Constructeurs

Écrire les constructeurs `public Matrix(int width, int height)` et `public Matrix()`. Le constructeur `Matrix(int width, int height)` initialise une matrice de largeur `width` et de largeur `height`, telle que toutes ses valeurs sont initialisées à 0. Le constructeur `public Matrix()` initialise une matrice vide.

2.3 Ajout d'un vecteur

Écrire la fonction `public void addElement(Vector vector)` qui ajoute une colonne à la matrice, augmentant sa largeur de 1. Si la matrice n'est pas vide, il faut vérifier que la nouvelle colonne est de même dimension que les colonnes déjà présentes. Dans le cas contraire, la fonction `addElement` renvoie une exception de type `InvalidParameterException`.

2.4 Accesseurs

Écrire les méthodes `public int width()`, `public int height()`, `public Vector getElement(int i)` et `public double getElement(int i, int j)`. La méthode `width()` renvoie la largeur de la matrice, `height()` renvoie la hauteur de la matrice, et `getElement(i)` renvoie $M[i]$.

2.5 Somme de deux matrices

Écrire la méthode `public Matrix sum(Matrix matrix)` qui réalise la somme de deux matrices de mêmes dimensions. Cette méthode crée une nouvelle matrice. Si les deux matrices à additionner n'ont pas les même

dimensions, la méthode `sum` doit lancer une exception de type `InvalidParameterException`. Pour M, M' deux matrices, la somme donnée par $M.sum(M')$ est une matrice qui vérifie $M.sum(M')[i,j] = M[i,j] + M'[i,j]$, pour i, j toute paire d'indices valide pour M.

2.6 Produit d'une matrice par un vecteur

Écrire la méthode `public Vector product(Vector vector)` qui renvoie le produit de la matrice par le vecteur `vector`. Cette méthode crée un nouveau vecteur. Étant donné une matrice M de dimensions $m \times n$ et un vecteur $\mathbf{v}$ de dimension n, le produit de M par $\mathbf{v}$ est le vecteur $\mathbf{u}$ de dimension m défini de la façon suivante :

$$\mathbf{u} = \sum_{i=0}^{n-1} (\mathbf{v}[i] \times M[i]),$$

où $\mathbf{v}[i] \times M[i]$ est le produit entre le i-ème élément de $\mathbf{v}$ (un scalaire) et la i-ème colonne de M (un vecteur). Si la matrice et le vecteur à multiplier n'ont pas des dimensions compatibles, la méthode `product(Vector vector)` doit lancer une exception de type `InvalidParameterException`.

2.7 Produit matriciel

Écrire la méthode `public Matrix product(Matrix matrix)` qui réalise le produit d'une matrice de dimensions $n \times m$ et d'une matrice de dimensions $o \times n$. Cette méthode crée une nouvelle matrice de dimensions $o \times m$. Si les deux matrices à multiplier n'ont pas des dimensions compatibles, la méthode `product(Matrix matrix)` doit lancer une exception de type `InvalidParameterException`. Pour M, M' deux matrices, le produit donnée par $M.product(M')$ est une matrice qui vérifie $M.product(M')[i] = M \times M'[i]$, pour i tout indice de colonne valide pour M'.

3 Tenseurs

Un *tenseur* est un type générique qui permet de représenter des scalaires (tenseurs d'ordre 0), des vecteurs (tenseurs d'ordre 1) et également des matrices (tenseurs d'ordre 2). On crée une interface `Tensor` pour abstraire les méthodes similaires des classes `Vector` et `Matrix` ; c'est-à-dire que ces classes vont implementer l'interface `Tensor`. L'interface `Tensor` est définie de la manière suivante :

```
public interface Tensor{
    public enum Type {SCALAR, VECTOR, MATRIX}
    public Tensor.Type getType();
    public int dim();
    public Tensor getElt(int index) throws IndexOutOfBoundsException;
    public Tensor product(Tensor other);
    public Tensor sum(Tensor other) throws IncompatibleDimensionsException;
}
```

On crée une nouvelle classe `Scalar` pour représenter des scalaires, qui implémente l'interface `Tensor`.

3.1 Déclaration de la classe Scalar

Écrire le code de la classe `Scalar` qui implemente l'interface `Tensor`, en déclarant la classe et en indiquant ses propriétés.

3.2 Produit d'un scalaire par un Tenseur

Écrire la méthode `public Tensor product(Tensor other)` de la classe `Scalar`, qui réalise le produit du scalaire avec le tenseur `other` passé en paramètre (qui peut être un tenseur de type `SCALAR`, `VECTOR` ou `MATRIX`). Selon le type du tenseur `other`, vous pouvez faire des appels aux méthodes existantes dans les classes `Vector` ou `Matrix`. La méthode doit renvoyer un objet de type `Tensor`.

3.3 Somme d'un scalaire et d'un Tenseur

Écrire la méthode `public Tensor sum(Tensor other)` de la classe `Scalar`, qui réalise la somme du scalaire avec le tenseur `other` passé en paramètre (qui peut être un tenseur de type `SCALAR`, `VECTOR` ou `MATRIX`). La somme n'est définie que si `other` est un objet de la classe `Scalar`. Si la somme ne peut être effectuée, la méthode lance une exception du type `IncompatibleDimensionsException`, sinon la méthode renvoie un objet de type `Tensor`.

ANNEXE

Bibliothèque `java.util.ArrayList<E>`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

boolean         add(E e);
// Appends the specified element to the end of this list.

void         add(int index, E element);
// Inserts the specified element at the specified position in this list.

E         get(int index);
// Returns the element at the specified position in this list.

boolean         isEmpty();
// Returns true if this list contains no elements.

Iterator<E>         iterator();
// Returns an iterator over the elements in this list in proper sequence.

E         remove(int index);
// Removes the element at the specified position in this list.

E         set(int index, E element);
// Replaces the element at the specified position in this list with the specified element.

int         size();
// Returns the number of elements in this list.
```