Pratique de la programmation orientée-objet
Examen intermédiaire

13 avril 2022

Indications :

— l'examen dure de 13h15 à 15h00,

— indiquez votre nom, prénom et numéro SCIPER ci-dessous et sur toutes les éventuelles feuilles additionnelles que vous rendriez,

— placez votre carte d'étudiant sur la table.

Seuls les documents suivants sont autorisés, en version papier uniquement :

— les notes de cours,

— les énoncés et les corrigés des séries d'exercices,

— une feuille de résumé de format A4 au maximum, ne couvrant que le cours.

Aucun document concernant le projet n'est autorisé !

Bon travail !

Nom : _____

Prénom : _____

SCIPER : _____

1 Décorateurs de fonctions [15 points]

Dans le projet, nous avons représenté les profils comme des fonctions qui, étant donné une position le long du profil, retournent l'altitude à cette position. Pour représenter le type de ces fonctions, nous avons utilisé l'interface `DoubleUnaryOperator`, définie comme suit :

```
public interface DoubleUnaryOperator {
  double applyAsDouble(double x);
}
```

Par exemple, la fonction $f(x) = x^3$ pourrait être représentée à l'aide d'une instance de la classe `Cubed` suivante :

```
public final class Cubed implements DoubleUnaryOperator {
  @Override public double applyAsDouble(double x) { return x * x * x; }
}
```

Le but de cet exercice est de définir et d'utiliser des décorateurs pour des fonctions représentées par des valeurs de type `DoubleUnaryOperator`.

Partie 1 [5 points] Écrivez la définition d'un enregistrement nommé `FlippedH`, un décorateur qui applique une symétrie horizontale à une fonction, autour de l'axe des y.

Le test JUnit suivant, qui doit s'exécuter avec succès, montre comment il pourrait être utilisé pour obtenir la fonction $g(x) = (-x)^3$ en utilisant une instance de `Cubed` :

```
DoubleUnaryOperator flippedCubed = new FlippedH(new Cubed());
assertEquals(-8.0, flippedCubed.applyAsDouble(2));
```

Réponse :

Partie 2 [5 points] Écrivez la définition d'un enregistrement nommé `Translated`, un décorateur qui translate une fonction d'une distance donnée le long de l'axe des x.

Le test JUnit suivant, qui doit s'exécuter avec succès, montre comment il pourrait être utilisé pour obtenir la fonction $f(x) = (x-1)^3$ en utilisant une instance de `Cubed` :

```
DoubleUnaryOperator translatedCubed = new Translated(new Cubed(), 1);
assertEquals(-8, translatedCubed.applyAsDouble(-1));
```

Remarquez que la valeur passée au constructeur de `Translated` donne la distance par laquelle la fonction doit être translatée le long de l'axe des x. Par conséquent, une valeur positive la déplace vers la droite, tandis qu'une valeur négative la déplace vers la gauche.

Réponse :

Partie 3 [5 points] En utilisant les décorateurs que vous avez définis, écrivez une méthode nommée `invertedEdgeProfile` qui, étant donné une arête de type `Edge`, retourne son profil inversé, de type `DoubleUnaryOperator`. Le profil inversé d'une arête est celui qui correspond à un parcours de l'arête en sens inverse.

Pour mémoire, parmi les méthodes offertes par la classe `Edge` figurent les deux suivantes, qui pourraient être utiles ici :

— `DoubleUnaryOperator profile()`, qui retourne le profil de l'arête,

— **`double`** `length()`, qui retourne la longueur de l'arête.

Réponse :

2 Itérateur indexé [25 points]

Lorsqu'on itère sur les éléments d'une collection à l'aide d'un itérateur, il est parfois utile de connaître l'index de l'élément courant. Le but de cet exercice est d'écrire une classe représentant un itérateur sur des « éléments indexés », c.-à-d. des éléments accompagnés de leur index.

Pour cela, on définit tout d'abord un enregistrement associant un élément à son index :

```
public record Indexed<E>(int index, E element) {}
```

En l'utilisant, il est possible d'écrire une classe d'itérateur nommée `IndexedIterator` qui, étant donné un itérateur sur certains éléments, itère sur ces mêmes éléments mais accompagnés de leur index. L'extrait de programme suivant illustre son utilisation :

```
List<String> l = List.of("Javascript", "Python", "Java");
Iterator<Indexed<String>> it = new IndexedIterator<>(l.iterator());
while (it.hasNext()) {
  Indexed<String> i = it.next();
  System.out.print(i.index() + ": " + i.element() + ", ");
}
```

L'exécution de cet extrait de programme devrait afficher la ligne suivante :

```
0: Javascript, 1: Python, 2: Java,
```

Partie 1 [3 points] Écrivez la déclaration de la classe `IndexedIterator`, qui est générique et implémente l'interface `Iterator`. N'écrivez que la définition de la classe pour l'instant, n'y ajoutez aucun attribut, constructeur ou méthode.

Réponse :

Partie 2 [4 points] Écrivez le ou les attributs de la classe `IndexedIterator`, ainsi que son constructeur. Comme illustré dans l'exemple ci-dessus, le constructeur prend un seul argument, l'itérateur produisant les éléments à indexer.

Réponse :

Partie 3 [4 points] Écrivez les définitions des méthodes `hasNext` et `next`.

Réponse :

Partie 4 [4 points] Écrivez la définition de la méthode `remove`, qui supprime le dernier élément retourné par `next` et met à jour l'index en conséquence. Le test JUnit suivant, qui doit s'exécuter avec succès, illustre son comportement :

```
List<String> l = new ArrayList<>(List.of("a", "b", "c"));
Iterator<Indexed<String>> it = new IndexedIterator<>(l.iterator());
it.next(); it.remove();
assertEquals(new Indexed<>(0, "b"), it.next());
assertEquals(List.of("b", "c"), l);
```

Réponse :

Partie 5 [4 points] Écrivez la définition d'une méthode statique et générique nommée `over` pour votre classe `IndexedIterator`, prenant un « itérable » et retournant un itérateur indexé pour ses éléments. Elle devrait être utilisable comme suit :

```
List<String> l = List.of("a", "b", "c");
Iterator<Indexed<String>> it = IndexedIterator.over(l);
```

Pour mémoire, un « itérable » est une instance d'une classe qui implémente l'interface `Iterable`, dont la définition est donnée en annexe.

Réponse :

Partie 6 [6 points] Utilisez votre classe `IndexedIterator` pour écrire une méthode générique nommée `indexOf` qui, étant donné une liste et un élément, retourne l'index de la première occurrence de l'élément dans la liste, ou -1 si l'élément n'y figure pas. Le test JUnit suivant, qui doit s'exécuter avec succès, illustre son utilisation :

```
List<String> l = List.of("Javascript", "Python", "Java");
assertEquals(2, indexOf(l, "Java"));
assertEquals(-1, indexOf(l, "Rust"));
```

Utilisez la méthode `equals` pour vérifier si un élément de la liste est égal à celui recherché.

Réponse :

3 Tableaux creux [35 points]

L'algorithme A* utilise un tableau de type `float[]` pour stocker la distance du plus court chemin connu vers chaque nœud du graphe. Les éléments de ce tableau sont initialisés à $+\infty$, et sont ensuite mis à jour au fur et à mesure que le graphe est visité. Cependant, dans la plupart des cas, l'algorithme ne visite pas tous les nœuds du graphe, et de nombreux éléments du tableau conservent donc leur valeur initiale. La mémoire utilisée pour les stocker est gaspillée.

Le but de cet exercice est de résoudre (partiellement) ce problème en écrivant une classe permettant de représenter efficacement un **tableau creux** (*sparse array*), c.-à-d. un tableau dont de nombreux éléments ne sont jamais modifiés et conservent leur valeur initiale, ici $+\infty$.

L'idée est la suivante : au lieu de stocker toutes les valeurs dans un seul grand tableau, on les stocke dans une séquence de tableaux plus petits appelés **morceaux** (*chunks*). Chaque morceau contient exactement 4096 (2^{12}) valeurs, sauf le dernier, qui peut en contenir moins. Le premier morceau contient les éléments d'index 0 à 4095, le second les éléments d'index 4096 à 8191, et ainsi de suite. Ces morceaux sont stockés dans un **tableau de morceaux** (*chunk array*). Pour économiser de la mémoire, un morceau n'est créé qu'au moment où une valeur différente de la valeur initiale y est stockée.

La classe `SparseFloatArray` ci-dessous, à compléter, représente un tableau creux de valeurs de type `float` en utilisant cette technique. Tout comme un tableau Java de type `float[]`, sa taille est fixée lors de sa création et ne change plus par la suite.

```
public final class SparseFloatArray {
  // à faire : attributs
  public SparseFloatArray(int size, float initialValue) { /* à faire */ }
  public int size() { /* à faire */ }
  public float get(int i) { /* à faire */ }
  public void set(int i, float value) { /* à faire */ }
  public double density() { /* à faire */ }
  @Override public String toString() { /* à faire */ }
}
```

Partie 1 [8 points] Écrivez les attributs de `SparseFloatArray` et le corps de son constructeur, qui doit lever une `IllegalArgumentException` si la taille qu'on lui passe est négative.

La valeur initiale donnée au constructeur est celle de tous les éléments du tableau. Donc la méthode `get` retourne la valeur initiale lorsqu'on l'appelle sur un tableau creux à peine créé.

Votre constructeur doit créer le tableau de morceaux lui-même, mais *aucun* des morceaux ! Les morceaux eux-mêmes doivent avoir le type `float[]` et pour calculer la taille du tableau qui les contient, vous pouvez supposer l'existence d'une méthode `ceilDiv` similaire à celle du projet, prenant deux entiers x et y et retournant l'entier $\lceil x/y \rceil$.

Réponse :

Réponse (suite) :

Partie 2 [2 points] Écrivez le corps de la méthode `size`, qui retourne la taille du tableau.

Réponse :

Partie 3 [5 points] Écrivez le corps de la méthode `get`, qui retourne l'élément du tableau à l'index donné, ou lève une `IndexOutOfBoundsException` si cet index n'est pas valide, c.-à-d. plus petit que 0 ou plus grand ou égal à la taille du tableau.

Votre méthode `get` ne doit en aucun cas créer un morceau.

Réponse :

Partie 4 [8 points] Écrivez le corps de la méthode `set`, qui stocke la valeur donnée à l'index donné dans le tableau, ou lève une `IndexOutOfBoundsException` si cet index est invalide.

Votre mise en œuvre ne doit créer un morceau qu'en cas de nécessité absolue. Cela signifie qu'un morceau dont tous les éléments sont égaux à la valeur initiale ne doit pas être créé. Cependant, un morceau déjà existant qui, après plusieurs modifications, finit par avoir tous ses éléments égaux à la valeur initiale ne doit pas être supprimé du tableau de morceaux.

Réponse :

Partie 5 [4 points] Écrivez le corps de la méthode `toString`, qui retourne une chaîne de caractères constituée des représentations textuelles de tous les éléments du tableau, séparées par des virgules et entourées de crochets. Utilisez la méthode `String.valueOf` pour obtenir la représentation textuelle d'un élément de type **`float`**.

Le test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre le comportement de la méthode `toString` :

```
SparseFloatArray a = new SparseFloatArray(2, (float) 3.14);
assertEquals("[3.14,3.14]", a.toString());
```

Réponse :

Réponse (suite) :

Partie 6 [8 points] Écrivez le corps de la méthode `density`, qui retourne la densité du tableau. Celle-ci est définie comme le rapport entre le nombre d'éléments du tableau différents de la valeur initiale et le nombre total d'éléments contenus dans les morceaux existants. Par convention, la densité vaut 1 si aucun morceau n'existe.

Réponse :

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque Java nécessaires à cet examen. De nombreuses méthodes ont été omises et les types parfois simplifiés.

Classe `Arrays`

La classe `java.util.Arrays`, non instanciable, contient des méthodes statiques de manipulation de tableaux.

```
public class Arrays {
  // Remplit la totalité du tableau a avec la valeur f donnée.
  static void fill(float[] a, float f);
}
```

Interface `Iterator`

L'interface `java.lang.Iterator` représente un itérateur, c.-à-d. un objet permettant de parcourir les éléments d'une collection.

```
public interface Iterator<E> {
  // Retourne vrai ssi l'itérateur a encore un élément à fournir.
  boolean hasNext();

  // Retourne le prochain élément, ou lève NoSuchElementException s'il n'y en a plus.
  E next();

  // Supprime l'élément retourné par le dernier appel à next.
  // Lève IllegalStateException si next n'a pas encore été appelée, ou si remove
  // a déjà été appelée après le dernier appel à next.
  void remove();
}
```

Interface `Iterable`

L'interface `java.lang.Iterable` représente un objet itérable, c-à-d dont le contenu peut être parcouru au moyen d'un itérateur, ou de la boucle *for-each*.

```
interface Iterable<E> {
  // Retourne un itérateur sur les éléments de l'objet.
  Iterator<E> iterator();
}
```

(suite à la page suivante)

Classe `StringJoiner`

La classe `java.lang.StringJoiner` représente un «joigneur» de chaîne, qui construit des chaînes de caractères constituées d'une séquence de chaînes séparées par un délimiteur et entourées d'un préfixe et d'un suffixe.

```
public class StringJoiner {
  // Construit un « joigneur » de chaînes avec le préfixe p, le suffixe s et
  // le séparateur d.
  StringJoiner(String d, String p, String s);

  // Ajoute la chaîne s à la séquence.
  void add(String s);

  // Retourne la chaîne constituée de la concaténation du prefixe, des chaînes
  // ajoutées jusqu'à présent et séparées par le séparateur, et du suffixe.
  String toString();
}
```