Pratique de la programmation orientée-objet
Examen intermédiaire

5 avril 2017

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 Tables bidimensionnelles [25 points]

La classe `StringTable` ci-dessous représente une table bidimensionnelle immuable, dont les index commencent à une valeur quelconque (c-à-d pas nécessairement 0) :

```
public final class StringTable {
  attributs à ajouter

  public StringTable(List<String> elements,
                     int minX, int width,
                     int minY, int height) {
    à faire
  }
  public String get(int x, int y) { à faire }
}
```

Par exemple, la table bidimensionnelle suivante :

	−1	**0**	**1**
1	a	b	c
2	d	e	f

peut être créée ainsi :

```
StringTable t =
  new StringTable(Arrays.asList("a","b","c","d","e","f"),
                  -1, 3,
                  1, 2);
```

et son élément d'index $(-1,2)$ peut être extrait et imprimé ainsi :

```
System.out.println(t.get(-1, 2)); // imprime "d"
```

Partie 1 [3 points] Transformez la classe `StringTable` ci-dessus en une classe générique nommée `Table` et capable de stocker des éléments d'un type quelconque, pas seulement des chaînes.

Pour l'instant, contentez-vous de donner l'équivalent générique de la classe ci-dessus, sans définir d'attributs ou écrire le corps des méthodes.

Réponse :

Partie 2 [6 points] Écrivez les attributs de la classe générique `Table` ainsi que le corps de son constructeur.

Le constructeur doit lever `IllegalArgumentException` si la largeur ou la hauteur sont (strictement) négatifs, ou si la liste d'éléments donnée n'est pas de la bonne taille pour la table.

Souvenez-vous que la table doit être immuable !

Réponse :

Partie 3 [6 points] Écrivez le corps de la méthode `get`, qui retourne l'élément d'index donné ou lève `IndexOutOfBoundsException` si l'index est invalide.

Réponse :

Partie 4 [10 points] Ajoutez une redéfinition de la méthode `toString` à la classe `Table`, qui retourne une chaîne dans laquelle les éléments de la table apparaissent ligne par ligne, ordonnés de haut en bas et de gauche à droite. Les éléments d'une ligne et les lignes elles-mêmes doivent être séparés par une virgule, et chaque ligne ainsi que la table entière doivent être entourés de crochets.

Par exemple, appliquée à (l'équivalent générique de) la table `t` ci-dessus, la méthode `toString` doit retourner la chaîne suivante :

```
[[a, b, c], [d, e, f]]
```

Réponse :

2 Index de livre [40 points]

L'index d'un livre est une liste triée par ordre alphabétique des termes importants figurant dans le livre, accompagnés d'information à leur sujet. Dans le cadre de cet exercice, cette information peut être de deux types :

1. une *référence de page*, donnant une liste triée par ordre croissant des numéros des pages sur lesquelles le terme figure,

2. une *référence croisée* à un autre terme apparaissant dans l'index.

Par exemple, un index très simple contenant 4 entrées pour un livre sur la programmation pourrait ressembler à ceci :

> ensemble, 27, 28, 29
> liste, 23, 24, 25, 26
> table associative, 29
> tableau dynamique, *voir* liste

Dans cet exemple, les trois premières entrées sont des références de page, tandis que la dernière est une référence croisée.

En Java, un tel index peut être représenté au moyen de deux tables associatives : une contenant les références de page, l'autre contenant les références croisées. La classe immuable `Index` ci-dessous, à compléter, est basée sur cette idée.

```
public final class Index {
  private final Map<String, Set<Integer>> pageRefs;
  private final Map<String, String> crossRefs;

  public Index(Map<String, Set<Integer>> pageRefs,
               Map<String, String> crossRefs) { à faire }

  public List<String> toLines() { à faire }
  à compléter
}
```

Partie 1 [14 points] Écrivez le corps du constructeur de la classe, qui lève l'exception `NullPointerException` si l'un de ses arguments est `null`, ou l'exception `IllegalArgumentException` si au moins un des numéros de page n'est pas strictement positif (> 0), ou si une référence croisée désigne un terme n'apparaissant pas dans l'index.

Souvenez-vous que la classe doit être immuable et notez qu'une référence croisée peut désigner une autre référence croisée. Cela signifie, par exemple, qu'un index consistant en deux références croisées se référençant mutuellement, comme celui ci-dessous, est valide :

> boucle infinie, *voir* infinie, boucle
> infinie, boucle, *voir* boucle infinie

Réponse :

Partie 2 [12 points] Écrivez le corps de la méthode `toLines`, qui retourne une liste des lignes représentant l'index. Par exemple, appliquée à l'objet représentant l'index donné dans l'introduction, elle doit retourner une liste contenant les quatre chaînes suivantes, dans cet ordre (c-à-d triées alphabétiquement) :

```
ensemble, 27, 28, 29
liste, 23, 24, 25, 26
table associative, 29
tableau dynamique, voir liste
```

Réponse :

Partie 3 [14 points] La classe `Index` étant immuable, il est utile de fournir un bâtisseur pour en construire des instances de manière incrémentale. Dans ce but, écrivez une classe nommée `Builder`, imbriquée statiquement dans `Index` et offrant trois méthodes :

- `Builder add(String word, int page)`, qui ajoute au bâtisseur une référence de page et retourne `this`, ou lève `IllegalArgumentException` si le bâtisseur contient déjà une référence *croisée* pour le mot donné, ou si le numéro de page n'est pas strictement positif (> 0),
- `Builder add(String word, String reference)`, qui ajoute ou remplace une référence croisée au bâtisseur et retourne `this`, ou lève l'exception `IllegalArgumentException` si le bâtisseur contient déjà une référence *de page* pour le mot `word` donné (s'il contient déjà une référence *croisée* pour ce mot, elle est remplacée par la nouvelle),
- `Index build()`, qui retourne un index contenant toutes les références ajoutées au bâtisseur jusqu'à présent.

Notez que la seconde méthode `add` ne vérifie *pas* que le mot référencé existe déjà dans l'index en construction. Cette vérification est faite plus tard par le constructeur de `Index`, comme expliqué plus haut.

La classe `Builder` doit pouvoir s'utiliser ainsi pour construire l'index donné dans l'introduction :

```
Index.Builder ib = new Index.Builder()
  .add("tableau␣dynamique", "liste")
  .add("table␣associative", 29);
for (int p = 27; p <= 29; ++p)
  ib.add("ensemble", p);
for (int p = 23; p <= 26; ++p)
  ib.add("liste", p);
Index i = ib.build();
```

Réponse :

(suite à la page suivante)

Réponse (suite) :

3 Tableaux [10 points]

La classe non instanciable `DoubleArrays` ci-dessous fournit trois méthodes statiques permettant d'effectuer différentes opérations sur des tableaux d'éléments de type `double`. Le corps de ces méthodes a systématiquement été omis et est à écrire en suivant les explications données plus bas.

```
public final class DoubleArrays {
  private DoubleArrays() {}

  public static void forEach(double[] a,
                             DoubleConsumer c);
  public static void replaceAll(double[] a,
                                DoubleUnaryOperator op);
  public static double reduceLeft(double z,
                                  double[] a,
                                  DoubleBinaryOperator op);
}
```

Partie 1 [3 points] Écrivez le corps de la méthode `forEach`, qui applique le consommateur donné à tous les éléments du tableau, du premier au dernier. Par exemple, cette méthode pourrait être utilisée ainsi pour imprimer les éléments du tableau `a1`, un par ligne :

```
double[] a1 = new double[] { 1, 2, 3.14 };
DoubleArrays.forEach(a1, System.out::println);
```

Réponse :

Partie 2 [3 points] Écrivez le corps de la méthode `replaceAll`, qui remplace chaque élément du tableau par le résultat de l'application de l'opérateur unaire donné à cet élément. Par exemple, cette méthode pourrait être utilisée ainsi pour remplacer tous les éléments du tableau `a2` par leur carré, après quoi le tableau contiendrait 1, 4 et 9.

```
double[] a2 = new double[] { 1, 2, 3 };
DoubleArrays.replaceAll(a2, x -> x * x);
```

Réponse :

Partie 3 [4 points] Écrivez le corps de la méthode `reduceLeft`, qui réduit progressivement, au moyen de l'opérateur binaire donné, la valeur initiale donnée et les éléments du tableau afin d'obtenir une valeur finale unique, qu'elle retourne.

La réduction se fait depuis la gauche, en appliquant d'abord l'opérateur binaire `op` à la valeur initiale `z` et au premier élément du tableau `a`, puis en combinant cette valeur avec le second élément du tableau, et ainsi de suite.

Par exemple, `reduceLeft` peut être utilisée ainsi pour calculer le produit `p` de tous les éléments du tableau `a3`, qui vaut 120 :

```
double[] a3 = new double[] { 2, 3, 4, 5 };
double p = DoubleArrays.reduceLeft(1, a3, (x, y) -> x * y);
```

Cette réduction équivaut à calculer l'expression suivante :

$$((((1 \times 2) \times 3) \times 4) \times 5) = 120$$

Comme autre exemple, `reduceLeft` peut aussi être utilisée comme suit pour calculer la somme `s` de tous les éléments du tableau `a3`, qui vaut 14 :

```
double s = DoubleArrays.reduceLeft(0, a3, (x, y) -> x + y);
```

Cette réduction équivaut à calculer l'expression suivante :

$$((((0 + 2) + 3) + 4) + 5) = 14$$

Notez que `reduceLeft` retourne simplement la valeur initiale lorsqu'on l'applique à un tableau vide.

Réponse :

Formulaire

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

Classe `StringBuilder`

La classe `java.lang.StringBuilder` est un bâtisseur de chaîne de caractères.

```
public class StringBuilder {
  // Ajoute la représentation textuelle de l'objet o, fournie par sa méthode
  // toString, à la chaîne en cours de construction. Retourne this.
  public StringBuilder append(Object o);

  // Retourne une nouvelle chaîne avec le contenu ajouté jusqu'à présent.
  public String toString();
}
```

Interface `Collection`

L'interface `java.util.Collection` représente une collection de valeurs. Elle est implémentée, entre autres, par les classes `ArrayList`, `HashSet` et `TreeSet`. Toutes ces classes offrent un constructeur de copie prenant une collection en argument.

```
public interface Collection<E> extends Iterable<E> {
  // Retourne la taille (nombre d'éléments) de la collection.
  public int size();

  // Ajoute l'élément e à la collection.
  public boolean add(E e);

  // Ajoute tous les éléments de c à la collection.
  public boolean addAll(Collection<E> c);

  // Supprime tous les éléments de c de la collection.
  public boolean removeAll(Collection<E> c);

  // Ne garde dans la collection que les éléments se trouvant également dans c.
  public boolean retainAll(Collection<E> c);
}
```

Interface `List`

L'interface `java.util.List` représente les listes. Elle est implémentée, entre autres, par la classes `ArrayList`.

```
public interface List<E> extends Collection<E> {
  // Retourne l'élément d'index i.
  E get(int i);
}
```

Interface `Map`

L'interface `java.util.Map` représente les tables associatives. Elle est implémentée, entre autres, par les classes `HashMap` et `TreeMap`. Ces classes possèdent un constructeur de copie prenant une table associative en argument.

```
public interface Map<K, V> {
  // Retourne vrai ssi la table contient la clef k.
  public boolean containsKey(K k);

  // Associe la valeur v à la clef k.
  public V put(K k, V v);

  // Retourne la valeur associée à k, ou null s'il n'y en a aucune.
  public V get(K k);

  // Retourne une vue sur l'ensemble des clefs.
  public Set<K> keySet();

  // Retourne une vue sur la collection des valeurs.
  public Collection<V> values();
}
```

Classe `Collections`

La classe `java.util.Collections`, non instanciable, contient des méthodes statiques travaillant sur les collections.

```
public class Collections {
  // Retourne une vue non modifiable sur la liste l.
  public static <T> List<T> unmodifiableList(List<T> l);

  // Retourne une vue non modifiable sur l'ensemble s.
  public static <T> Set<T> unmodifiableSet(Set<T> s);

  // Retourne une vue non modifiable sur la table associative m.
  public static <K, V> Map<K, V> unmodifiableMap(Map<K, V> m);
}
```

Interfaces fonctionnelles

Les interfaces fonctionnelles du paquetage `java.util.function` représentent différents types de fonctions. Seule leur (unique) méthode abstraite est donnée ici.

```
public interface DoubleConsumer {
  public void accept(double value);
}
public interface DoubleUnaryOperator {
  public double applyAsDouble(double operand);
}
public interface DoubleBinaryOperator {
  public double applyAsDouble(double left, double right);
}
```