Pratique de la programmation orientée-objet
Examen final

2 juin 2017

Indications :

- l'examen dure de 12h15 à 16h00,
- 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 Encodage UTF-8 [35 points]

UTF-8 est un encodage d'Unicode à taille variable, qui représente un caractère au moyen de 1 à 4 octets, en fonction de son point de code. L'encodage d'un caractère est fait de la manière suivante, où b_i représente le i-ème bit de son point de code, b_0 étant le bit de poids le plus faible :

- un caractère dont le point de code est compris entre 0 et $7F_{16}$ est encodé par un unique octet égal au point de code, c-à-d avec son bit de poids fort mis à 0 : $0b_6b_5b_4b_3b_2b_1b_0$,
- un caractère dont le point de code est compris entre 80_{16} et $7FF_{16}$ est encodé par deux octets :
  1. le premier contient les 5 bits de poids fort du point de code, précédés de la séquence de trois bits 110_2 : $110b_{10}b_9b_8b_7b_6$,
  2. le second contient les 6 bits de poids faible du point de code, précédés de la séquence de deux bits 10_2 : $10b_5b_4b_3b_2b_1b_0$,
- un caractère dont le point de code est compris entre 800_{16} et $FFFF_{16}$ est encodé par trois octets :
  1. le premier contient les 4 bits de poids fort du point de code, précédés de la séquence de 4 bits 1110_2 : $1110b_{15}b_{14}b_{13}b_{12}$,
  2. le second contient les 6 bits de poids moyen, précédés de la séquence de 2 bits 10_2 : $10b_{11}b_{10}b_9b_8b_7b_6$,
  3. le troisième contient les 6 bits de poids faible, précédés de la séquence de 2 bits 10_2 : $10b_5b_4b_3b_2b_1b_0$,
- un caractère dont le point de code est supérieur à $FFFF_{16}$ est encodé par quatre octets, mais nous ignorerons ces caractères ici.

Par exemple, le caractère œ a le point de code 153_{16}, et est dès lors encodé par deux octets. Comme 153_{16} est 101010011_2, le premier octet vaut 11000101_2 ($C5_{16}$) et le second 10010011_2 (93_{16}).

Le but de cet exercice est d'écrire la classe `UTF8OutputStreamWriter`, un écrivain qui encode les caractères qu'on lui fournit au moyen de UTF-8 et écrit les octets résultants dans un flot d'octets sous-jacent. Il doit être possible d'utiliser cette classe comme l'extrait de test JUnit suivant l'illustre :

```
ByteArrayOutputStream bs = new ByteArrayOutputStream();
try (Writer w = new UTF8OutputStreamWriter(bs)) {
  w.write("œuf␣à␣€1");
}
byte[] expected = new byte[]{
  (byte)0xC5, (byte)0x93, (byte)0x75, (byte)0x66,
  (byte)0x20, (byte)0xC3, (byte)0xA0, (byte)0x20,
  (byte)0xE2, (byte)0x82, (byte)0xAC, (byte)0x31
};
assertArrayEquals(expected, bs.toByteArray());
```

Partie 1 [1 point] Nommez le patron de conception qui est utilisé par la classe `UTF8OutputStreamWriter` : _____ .

Partie 2 [34 points] Écrivez la classe instanciable `UTF8OutputStreamWriter`. Bien entendu, vous n'avez *pas* le droit d'utiliser la classe `OutputStreamWriter` de Java dans votre mise en œuvre !

Votre classe doit offrir un constructeur utilisable comme illustré dans le test JUnit ci-dessus, et ne doit redéfinir que les deux méthodes suivantes de `Writer` :

- `void write(char[] chars, int offset, int length)`, qui écrit la portion de `chars` débutant à la position `offset` et de longueur `length`,
- `void close()`, qui ferme l'écrivain.

Souvenez-vous qu'en Java, une valeur de type `char` est un entier dont la valeur est le point de code du caractère Unicode qu'elle représente, et cet entier est toujours compris entre 0 et $FFFF_{16}$. Les caractères Unicode dont le point de code est supérieur à $FFFF_{16}$ ne sont pas représentables au moyen d'une seule valeur de type `char` en Java, raison pour laquelle nous les ignorons ici.

Réponse :

(suite au verso)

Réponse (suite) :

2 Dessin de panorama [10 points]

Soit le panorama défini par les paramètres suivants :

Paramètre	**Valeur**
Position de l'observateur (longitude, latitude)	7° E, 46° N
Altitude de l'observateur	4000 m
Azimut central	0°
Angle de vue horizontal	90°
Distance maximale de visibilité	400 km
Dimensions (largeur × hauteur)	10 × 7

Partie 1 [4 points] La figure 1 représente un profil altimétrique utilisé lors du calcul du panorama ci-dessus. Les lignes traitillées représentent les angles d'élévation compris entre $-40°$ et $+40°$ et sont séparées de 5°. L'observateur se trouve à l'intersection de ces lignes, à l'endroit indiqué par le petit disque.

Sur cette figure, dessinez les rayons (partiels) lancés lors du calcul *optimisé* de la portion du panorama correspondant à ce profil altimétrique.

Notez que nous faisons l'hypothèse que la zone grisée représente le terrain ajusté, c-à-d celui prenant en compte la courbure terrestre et la réfraction. Dès lors, les rayons se déplacent en ligne droite sur cette figure.

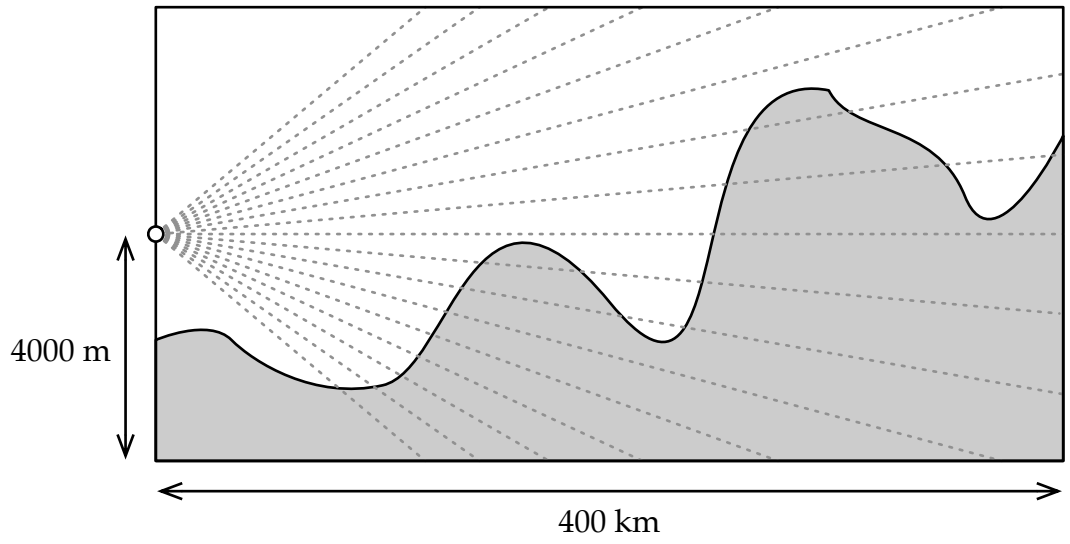

Fig. 1: Profil altimétrique

Partie 2 [3 points] Sur la carte de la figure 2, dessinez la zone visible du panorama. Souvenez-vous qu'un degré de longitude ou latitude fait au plus 111 km.

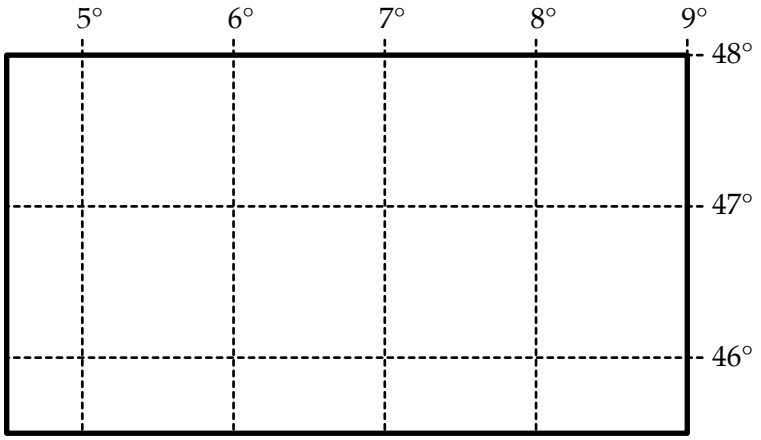

Fig. 2: Carte de la zone visible du panorama

Partie 3 [3 points] La figure 3 représente l'image du panorama en l'absence de suréchantillonnage. Les 10 lignes traitillées verticales et les 7 horizontales indiquent la position des échantillons : il y en a un à chaque intersection de deux lignes. Au sommet et sur la droite de l'image figurent des marques plus épaisses, que vous devez annoter ainsi :

- au-dessus de chacune des 10 marques se trouvant au sommet de l'image, écrivez l'azimut *canonique* correspondant, en degrés,
- à droite de chacune des 7 marques se trouvant à droite de l'image, écrivez l'angle d'élévation correspondant, en degrés.

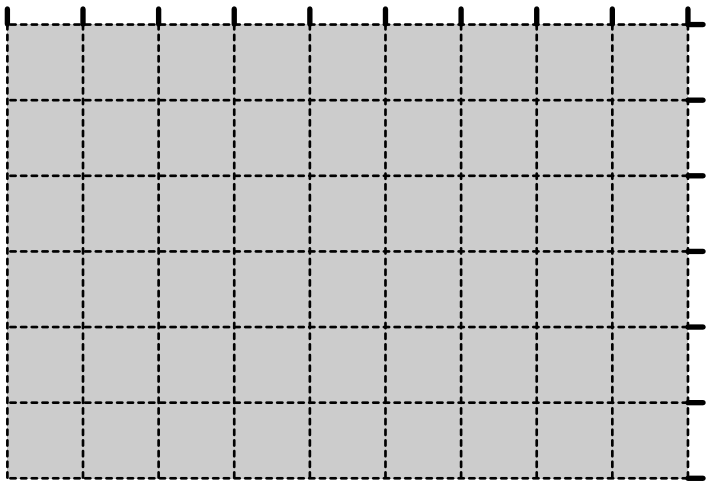

Fig. 3: Image du panorama

3 Cordes [42 points]

En Java, la classe `String` est, à juste titre, immuable. Dès lors, pour concaténer deux chaînes de caractères, il est nécessaire de créer un bâtisseur de chaîne, de lui ajouter les deux chaînes et enfin de lui faire construire la chaîne résultante.

Une solution alternative pour concaténer des chaînes immuables consiste à représenter la concaténation de deux chaînes comme un objet contenant les chaînes à concaténer. De manière similaire, il est possible de représenter l'extraction d'une portion d'une chaîne au moyen d'une vue, sans créer effectivement une nouvelle chaîne.

Le but de cet exercice est d'explorer une représentation alternative des chaînes de caractères basée sur ces idées. En informatique, ce type de chaîne est connu sous le nom de *corde* (*rope* en anglais), terminologie que nous adoptons ici.

L'interface `Rope` ci-dessous représente le concept de corde. Elle utilise les classes `StringRope`, `SubRope` et `ConcatRope` que vous devrez écrire.

```
public interface Rope {
  public static Rope ofString(String s) {
    return new StringRope(s);
  }
  public default Rope substring(int offset, int length) {
    return new SubRope(this, offset, length);
  }
  public default Rope append(Rope that) {
    return new ConcatRope(this, that);
  }
  public default void writeTo(Writer w) throws IOException {
    writeTo(w, 0, length());
  }
  public int length();
  public char charAt(int i);
  public void writeTo(Writer w, int offset, int length)
    throws IOException;
}
```

L'exemple suivant utilise cette interface pour créer une corde qui est une portion d'une concaténation de deux cordes, et l'imprime à l'écran :

```
Rope s = Rope.ofString("anticonstitutionnel")
  .append(Rope.ofString("lement"))
  .substring(4, 17);
Writer w = new OutputStreamWriter(System.out);
s.writeTo(w);  // imprime « constitutionnelle »
```

Partie 1 [3 points] Nommez les patrons de conception utilisés par les classes :

1. `StringRope` : _____ ,
2. `SubRope` : _____ ,
3. `ConcatRope` : _____ .

Partie 2 [13 points] Écrivez la classe instanciable `StringRope`, implémentant l'interface `Rope` et représentant une corde constituée d'une unique chaîne Java.

Son constructeur doit être utilisable comme illustré par la méthode `ofString` de `Rope`. En plus du constructeur, elle doit bien entendu redéfinir les trois méthodes abstraites de `Rope`, à savoir :

1. `int length()`, qui retourne la longueur de la corde,
2. `char charAt(int i)`, qui retourne le caractère d'index `i` de la corde, ou lève `IndexOutOfBoundsException` si cet index est invalide,
3. `void writeTo(Writer w, int offset, int length)`, qui écrit dans l'écrivain `w` la portion de la corde débutant à la position `offset` et comportant `length` caractères, ou lève `IndexOutOfBoundsException` si `offset` et `length` ne désignent pas une portion valide de la corde.

Pour des raisons d'efficacité, votre mise en œuvre de `writeTo` ne doit contenir aucun appel à la méthode `charAt`.

Réponse :

Partie 3 [13 points] Écrivez la classe instanciable `SubRope`, implémentant l'interface `Rope` et représentant une portion d'une corde, déterminée par son index de départ (*offset* en anglais) et sa longueur.

Son constructeur doit être utilisable comme illustré par la méthode `substring` de `Rope` et doit lever l'exception `IndexOutOfBoundsException` si les arguments qui lui sont passés ne désignent pas une portion valide de la corde.

Pour des raisons d'efficacité, votre mise en œuvre de `writeTo` ne doit contenir aucun appel à la méthode `charAt`.

Réponse :

Partie 4 [13 points] Écrivez la classe instanciable `ConcatRope`, implémentant l'interface `Rope` et représentant la concaténation de deux cordes.

Son constructeur doit être utilisable comme illustré par la méthode `append` de `Rope`.

Pour des raisons d'efficacité, votre mise en œuvre de `writeTo` ne doit contenir aucun appel à la méthode `charAt`.

Réponse :

4 Ensembles d'entiers [38 points]

Dans le projet, nous avons développé la classe immuable `Interval1D`, représentant un intervalle d'entiers unidimensionnel et non vide. Sa définition, ainsi que ses méthodes les plus importantes, dont le code a été omis, sont rappelées ci-dessous :

```
public final class Interval1D {
  public Interval1D(int includedFrom, int includedTo) { … }

  public int includedFrom() { … }
  public int includedTo() { … }

  public boolean contains(int v) { … }
  public int size() { … }

  public boolean isUnionableWith(Interval1D that) { … }
  public Interval1D union(Interval1D that) { … }
}
```

En plus des méthodes ci-dessus, la classe `Interval1D` redéfinit également `equals` et `hashCode` afin que les intervalles soient comparés par structure.

Bien qu'une instance de `Interval1D` représente un intervalle, et donc un ensemble d'entiers, il n'est bien entendu pas possible de représenter n'importe quel ensemble d'entiers au moyen d'un seul intervalle. Par exemple, ni l'ensemble vide $\varnothing$ ni l'ensemble de deux éléments $\{4, 9\}$ ne peuvent être représentés au moyen d'un intervalle. Il est toutefois possible de représenter des ensembles d'entiers arbitraires au moyen d'une *liste*, éventuellement vide, d'intervalles, et c'est le but de cet exercice.

En général, un ensemble d'entiers donné a un nombre infini de représentations sous forme de listes d'intervalles. Par exemple, l'ensemble $\{1, 2, 3, 7, 8, 10\}$ pourrait être représenté par les listes d'intervalles suivantes :

- [[1..3], [7..8], [10..10]], ou
- [[1..2], [3..3], [7..8], [10..10]], ou
- [[3..3], [1..2], [7..8], [7..7], [10..10], [8..8], [8..8]],
- etc.

Néanmoins, étant donné un ensemble d'entiers, on peut définir sa *représentation canonique* comme étant l'unique liste d'intervalles telle que :

- aucune paire d'intervalles n'est unionable,
- les intervalles sont triés, c-à-d que la borne supérieure d'un intervalle est toujours inférieure à la borne inférieure de son successeur.

Dans l'exemple plus haut, la première liste d'intervalles est la représentation canonique pour l'ensemble d'entiers. La seconde n'est pas canonique car les deux premiers intervalles sont unionables, et la troisième ne l'est pas non plus car plusieurs paires d'intervalles sont unionables, et ils ne sont pas triés.

Le but de la classe `Intervals` ci-dessous, à compléter, est de représenter un ensemble d'entiers au moyen d'une liste d'intervalles (canonique).

```
public final class Intervals {
  private final List<Interval1D> intervals;

  public Intervals(List<Interval1D> intervals) { à faire }
  public List<Interval1D> intervals() { à faire }
  public int size() { à faire }
  public boolean contains(int v) { à faire }
  public Intervals union(Intervals that) { à faire }
}
```

Partie 1 [8 points] Écrivez le constructeur de la classe `Intervals` et la méthode d'accès `intervals`. Le constructeur doit lever `IllegalArgumentException` si la liste d'intervalles donnée n'est pas canonique. Souvenez-vous que la classe doit être immuable !

Réponse :

Partie 2 [4 points] Écrivez la méthode `size`, qui retourne la taille de l'ensemble, c-à-d le nombre de ses éléments.

Par exemple, pour l'ensemble $\{1, 2, 3, 7, 8, 10\}$, elle doit retourner 6.

Réponse :

Partie 3 [6 points] Écrivez la méthode `contains`, qui retourne vrai si et seulement si l'ensemble auquel on l'applique contient l'élément donné.

Votre mise en œuvre doit tirer parti du fait que les intervalles sont triés pour retourner son résultat aussi rapidement que possible.

Réponse :

Partie 4 [14 points] Écrivez la méthode `union`, qui retourne l'union du récepteur et de son argument.

Par exemple, l'union des ensembles $\{1, 3, 8\}$ et $\{2, 4, 5, 6, 7\}$, représentés respectivement par les listes d'intervalles canoniques [[1..1], [3..3], [8..8]] et [[2..2], [4..7]], est l'ensemble $\{1, 2, 3, 4, 5, 6, 7, 8\}$, représenté par la liste d'intervalles canonique [[1..8]].

Réponse :

Partie 5 [6 points] À votre avis, la classe `Intervals` doit-elle redéfinir les méthodes `equals` et `hashCode` ? Si oui, écrivez leurs redéfinitions ; sinon, justifiez votre réponse.

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.

Interface `Iterable`

L'interface `java.lang.Iterable` représente les objets itérables, c-à-d ceux 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();
}
```

Interface `List`

L'interface `java.util.List` représente les listes. Elle est implémentée, entre autres, par les classes `ArrayList` et `LinkedList`, offrant toutes les deux un constructeur de copie prenant une liste en argument.

```
public interface List<E> extends Iterable<E> {
  // Retourne vrai ssi la liste est vide.
  public boolean isEmpty();

  // Retourne la taille (nombre d'éléments) de la liste.
  public int size();

  // Insère l'élément e à l'index i.
  void add(int i, E e);

  // Ajoute tous les éléments de l à la fin de la liste.
  public void addAll(List<E> l);

  // Retourne l'élément d'index i.
  E get(int i);

  // Supprime l'élément d'index i et le retourne.
  E remove(int i);
}
```

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);
}
```

Classe `Math`

La classe `Math`, non instanciable, contient des méthodes statiques représentant des fonctions mathématiques courantes.

```
public class Math {
  // Retournent le minimum/maximum de x et y.
  public static int min(int x, int y);
  public static int max(int x, int y);
}
```

Classe `Writer`

La classe `java.io.Writer` représente les écrivains, c-à-d les flots de caractères de sortie. Toutes les méthodes peuvent lever `IOException`.

```
abstract public class Writer {
  // Écrit la portion de la chaîne s débutant à l'index o et de longueur l.
  public void write(String s, int o, int l);

  // Écrit la portion du tableau c débutant à l'index o et de longueur l.
  public void write(char[] c, int o, int l);

  // Ferme le flot.
  public void close();
}
```

Classe `OutputStream`

La classe `java.io.OutputStream` représente les flots (d'octets) de sortie. Toutes les méthodes peuvent lever `IOException`.

```
abstract public class OutputStream {
  // Écrit la portion du tableau b débutant à l'index o et de longueur l.
  public void write(byte[] b, int o, int l);

  // Ferme le flot.
  public void close();
}
```

Classe `String`

La classe `java.lang.String` représente les chaînes de caractères.

```
public class String {
  // Retourne la longueur de la chaîne.
  public int length();

  // Retourne le caractère à l'index i ou lève IndexOutOfBoudsException
  // s'il est invalide.
  public char charAt(int i);
}
```