Pratique de la programmation orientée-objet
Examen final

2 juin 2023

Indications :

— l'examen dure de 13h15 à 17h00,

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

— placez votre carte d'étudiant·e 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 Tableaux d'octets immuables [25 points]

En Java, les tableaux d'octets de type `byte[]` ne sont pas immuables, ce qui est parfois ennuyant. Par exemple, dans le mini-projet dédié à la compression LZW, nous avons utilisé des listes immuables de type `List<Integer>` pour représenter les tableaux d'octets stockés dans le dictionnaire. Même si cette solution fonctionne, elle n'est pas satisfaisante, sachant que ces listes utilisent plus de mémoire et sont plus lentes que de simples tableaux.

Le but de cet exercice est donc d'écrire une classe représentant un tableau d'octets immuable, utilisable dans ce genre de situations.

Partie 1 [3 points] Écrivez la définition de `ByteArray`, une classe instanciable et immuable dotée d'un seul attribut, un tableau d'octets de type `byte[]`. Équipez cette classe d'un unique constructeur privé prenant en argument un tableau d'octets et le stockant sans le copier dans l'attribut.

Le constructeur ne copie pas le tableau qu'on lui passe pour des raisons de performances. Pour que la classe soit néanmoins immuable, tous les appelants du constructeur doivent donc garantir que le contenu du tableau passé ne changera jamais. Le constructeur étant privé, tous ces appelants se trouvent dans la classe elle-même ou dans une classe imbriquée, ce qui facilite la garantie de cette propriété.

Réponse :

Partie 2 [2 points] Ajoutez à votre classe une méthode statique nommée `of` prenant en argument un entier de type `int` et retournant une instance de la classe dont le tableau d'octets contient un seul octet, constitué des 8 bits de poids faible de cet argument.

Réponse :

Partie 3 [5 points] Ajoutez à votre classe des redéfinitions des méthodes `hashCode` et `equals` afin que les instances de `ByteArray` soient comparées par structure.

Notez que la classe `Arrays` de la bibliothèque Java possède plusieurs méthodes statiques qui rendent ces redéfinitions presque triviales. Consultez le formulaire en fin d'examen pour trouver ces méthodes !

Réponse :

Partie 4 [15 points] Ajoutez un bâtisseur à votre classe, sous la forme d'une classe imbriquée statiquement, nommée `Builder` et offrant les méthodes suivantes :

— `boolean isEmpty()`, qui retourne vrai si et seulement si le tableau en cours de construction est vide,

— `int size()`, qui retourne la taille du tableau en cours de construction,

— `Builder add(int b)`, qui ajoute les 8 bits de poids faible de `b` à la fin du tableau en cours de construction puis retourne le bâtisseur,

— `Builder removeLast()`, qui supprime le dernier élément du tableau en cours de construction puis retourne le bâtisseur, ou lève `IllegalStateException` si le tableau est vide,

— `Builder clear()`, qui vide le tableau en cours de construction puis retourne le bâtisseur,

— `ByteArray build()`, qui retourne un tableau d'octets immuable dont le contenu est identique à celui en cours de construction.

Votre bâtisseur doit impérativement représenter le tableau en cours de construction, qui est initialement vide, par une valeur de type `byte[]`. La méthode `add` doit redimensionner — par copie — ce tableau lorsque nécessaire, mais les méthodes `removeLast` et `clear` n'ont pas besoin de le faire.

Réponse :

2 Base de données indexée [25 points]

Dans le projet Javions, la base de données contenant les caractéristiques fixes des aéronefs est stockée dans une archive Zip contenant 256 fichiers CSV. Chaque fichier porte un nom débutant par deux chiffres hexadécimaux, et contient les données de tous les aéronefs dont l'adresse OACI se termine par ces deux chiffres.

Chacun de ces fichiers comporte une ligne par aéronef, et chaque ligne est constituée de 6 attributs, séparés les uns des autres par une virgule. Le premier d'entre eux est l'adresse OACI de l'aéronef, et le contenu des autres importe peu pour cet exercice.

Une autre manière d'organiser cette base de données d'aéronefs est de la stocker dans un unique fichier CSV contenant la totalité des données. Malheureusement, le fichier résultant est volumineux, et une recherche linéaire des caractéristiques d'un aéronef est donc lente.

Pour l'accélérer, il est possible de construire, au démarrage du programme, un index associant à chaque adresse OACI figurant dans la base de données, la position — dans le fichier — du premier caractère de la ligne qui lui correspond. Grâce à cet index, il est ensuite possible d'obtenir rapidement les données d'un aéronef dont on connaît l'adresse OACI. Il suffit en effet de consulter l'index pour obtenir la position du premier caractère de la ligne qui contient les données, ouvrir le fichier, ignorer tous les caractères précédant le premier de la ligne, et enfin lire la ligne.

Par exemple, imaginons que la base de données ne comporte que trois aéronefs, et que le fichier CSV la contenant soit constitué des lignes suivantes :

```
4B062D,HB-CIV,C182,CESSNA 182 Skylane,L1P,L
4B1814,HB-JDC,A20N,AIRBUS A-320neo,L2J,M
4B4407,HB-ZTV,AS50,AEROSPATIALE AS-350 Ecureuil,H1T,L
```

Le premier caractère de la première ligne est à la position 0 dans le fichier. Le premier caractère de la seconde ligne est quant à lui à la position 44, car la première ligne fait 43 caractères, et chaque fin de ligne est représentée au moyen d'un seul caractère. Finalement, le premier caractère de la troisième ligne est à la position 85. Dès lors, l'index associe la position 0 à l'adresse OACI 4B062D, la position 44 à l'adresse OACI 4B1814, et la position 85 à l'adresse OACI 4B4407.

Le but de cet exercice est de compléter la classe `AircraftDatabase` ci-dessous, qui utilise un index de ce type pour offrir un accès efficace à une base de données stockée dans un unique fichier CSV dont le nom est passé au constructeur.

```
public final class AircraftDatabase {
  private final String fileName;
  private final Map<Integer, Integer> index;

  private static Map<Integer, Integer> buildIndex(String fileName)
    throws IOException { /* à faire */ }

  public AircraftDatabase(String fileName) throws IOException {
    this.fileName = fileName;
    this.index = buildIndex(fileName);
  }

  public String get(int icaoAddress)
    throws IOException { /* à faire */ }
}
```

Notez que dans cette classe, les adresses OACI sont représentées par des entiers de type **`int`**, plutôt que par des instances d'une classe spécifique comme dans le projet. Pour mémoire, une adresse OACI n'est rien d'autre qu'un entier de 24 bits interprété de manière non signée et représenté en hexadécimal.

Partie 1 [15 points] Écrivez le corps de la méthode `buildIndex`, qui prend en argument le nom du fichier CSV contenant la base de données et retourne une table associative — l'index — qui associe à chaque adresse OACI présente dans ce fichier la position du premier caractère de la ligne correspondant à cette adresse.

Vous pouvez supposer que le fichier est encodé en UTF-8 et valide — c.-à-d. qu'il est composé de lignes ayant le format décrit plus haut — et laisser les éventuelles erreurs d'entrée/sortie se propager à l'appelant de la méthode. Vous devez toutefois vous assurer que le flot utilisé pour la lecture est toujours fermé, même en cas d'erreur.

Réponse :

Partie 2 [10 points] Écrivez le corps de la méthode `get`, qui prend en argument une adresse OACI et retourne soit la ligne de la base de données correspondant à cette adresse, soit **`null`** si elle ne figure pas dans la base de données.

Votre méthode doit impérativement utiliser l'index créé précédemment pour obtenir la position du premier caractère de la ligne correspondant à l'adresse OACI, puis utiliser la méthode `skip` du flot — décrite dans le formulaire en annexe — pour ignorer les caractères précédents.

Réponse :

3 Table associative d'entiers [25 points]

Dans l'exercice précédent, l'index de la base de données est représenté par une table associative de type `Map<Integer,Integer>`. Les clefs et les valeurs de cette table sont donc des instances de la classe `Integer`, qui occupent beaucoup plus de place en mémoire que de simples entiers de type **`int`**, sans pour autant contenir plus d'information. Lorsque la base de données contient de nombreux aéronefs — p. ex. plus de 400 000 comme dans le projet Javions —, l'index nécessite donc une importante quantité de mémoire.

Pour stocker de manière plus compacte une table associative dont les clefs et les valeurs sont des entiers de 32 bits, on peut représenter chaque paire (clef, valeur) par un entier de 64 bits dont la clef constitue les 32 bits de poids fort, et la valeur les 32 bits de poids faible. Ces entiers de 64 bits peuvent ensuite être placés dans un tableau trié en ordre croissant, dans lequel la recherche peut se faire par dichotomie.

Par exemple, l'index de l'exercice précédent pourrait être représenté par un tableau de type **`long`**`[]` contenant les trois éléments suivants — donnés ici en hexadécimal, avec un léger espacement entre les 32 bits de poids fort et ceux de poids faible pour simplifier la lecture :

`004B062D 00000000, 004B1814 0000002C, 004B4407 00000055`

(Note : `2C` est la représentation hexadécimale de 44, tandis que `55` est celle de 85.)

Le but de cet exercice est de compléter la définition de la classe immuable `IntToIntMap` ci-dessous, qui utilise cette idée pour représenter de manière compacte une table associative dont les clefs et les valeurs sont des entiers de 32 bits.

```
public final class IntToIntMap {
  private final long[] keysAndValues;

  private static long pack(int key, int value) { /* à faire */ }

  public static IntToIntMap ofMap(Map<Integer, Integer> map)
    { /* à faire */ }

  // Pré-conditions : keysAndValues est trié, jamais modifié.
  private IntToIntMap(long[] keysAndValues) {
    this.keysAndValues = keysAndValues;
  }

  public int getOrDefault(int key, int defaultValue) { /* à faire */ }
}
```

Partie 1 [3 points] Écrivez le corps de la méthode `pack`, qui prend en arguments une clef et une valeur, et retourne l'entier de type **`long`** dont les 32 bits de poids fort contiennent la clef, et les 32 de poids faible la valeur. Attention : la clef et la valeur peuvent être négatives !

Réponse :

Partie 2 [7 points] Écrivez le corps de la méthode `ofMap`, qui prend en argument une table associative de type `Map<Integer,Integer>` et retourne la table de type `IntToIntMap` équivalente.

Notez que le constructeur de `IntToIntMap` est privé, et exige que le tableau qu'on lui passe soit trié en ordre croissant, et jamais modifié par la suite, afin de garantir l'immuabilité de la classe. De plus, aucun des éléments ne doit avoir les mêmes 32 bits de poids fort qu'un autre.

Réponse :

Partie 3 [15 points] Écrivez le corps de la méthode `getOrDefault`, qui prend en arguments une clef et une valeur par défaut, et qui retourne la valeur associée à la clef si celle-ci est présente dans la table, ou la valeur par défaut sinon.

Votre méthode doit impérativement effectuer la recherche par dichotomie au moyen de la méthode `binarySearch` de `Arrays`, décrite dans le formulaire à la fin de l'examen.

Réponse :

4 Décompression d'échantillons [25 points]

La radio utilisée dans le projet produit chaque seconde 20 millions d'échantillons. Chacun de ces échantillons fait 12 bits mais est représenté au moyen de deux octets, et occupe donc 16 bits. Dès lors, la radio produit chaque seconde 40 mégaoctets de données, ce qui est considérable, surtout si on désire enregistrer ces données dans un fichier, ou les transmettre via Internet à un autre ordinateur.

Il est possible de réduire la taille de ces données en tirant parti de leurs caractéristiques. En particulier, il est fréquent que deux échantillons successifs aient des valeurs proches, auquel cas leur différence peut se représenter en utilisant moins de 12 bits. On peut donc imaginer représenter une séquence d'échantillons provenant de la radio au moyen d'une séquence de valeurs de 16 bits contenant chacune entre 1 et 3 différences. Les bits de poids fort de chaque valeur — soulignés ci-dessous — sont utilisés pour déterminer le contenu des bits restants, selon le schéma suivant :

Bits	**Signification**
`1`aaaaabbbbbccccc	Trois différences (`a`, `b`, `c`) de 5 bits chacune
`00`aaaaaaabbbbbbb	Deux différences (`a`, `b`) de 7 bits chacune
`0100`aaaaaaaaaaaa	Une différence (`a`) de 12 bits

Par exemple, la seconde ligne spécifie que lorsque les deux bits de poids fort de la valeur de 16 bit valent 0, alors les 14 bits restants contiennent deux différences de 7 bits chacune, la première (`a`) occupant les bits d'index 7 à 13, la seconde (`b`) les bits d'index 0 à 6.

Bien entendu, les différences peuvent être négatives, et les plages de bits nommées `a`, `b` et `c` dans la table ci-dessus doivent donc être interprétées en complément à deux. Ainsi, la valeur de 16 bits `00 0000011 1111111` représente la séquence de différences $[+3, -1]$.

Le but de cet exercice est de compléter la définition de la classe `SamplesDecompressor` ci-dessous, qui permet d'obtenir une séquence d'échantillons à partir de valeurs de 16 bits ayant le format décrit ci-dessus.

```
public final class SamplesDecompressor {
  private static boolean testBit(int value, int bitIndex)
    { /* à faire */ }
  private static int extractSInt(int value, int start, int size)
    { /* à faire */ }
  public static int decompress(int value, short[] samples, int index)
    { /* à faire */ }
}
```

Partie 1 [2 points] Écrivez le corps de la méthode `testBit`, qui retourne vrai si et seulement si le bit d'index `bitIndex` de `value` vaut 1. On suppose, sans le vérifier explcitement au début de la méthode, que $0 \leq$ `bitIndex` < 32.

Réponse :

Partie 2 [3 points] Écrivez le corps de la méthode `extractSInt`, qui extrait de `value` la plage de `size` bits commençant au bit d'index `start`, qu'elle interprète comme une valeur signée en complément à deux. (Suggestion : pour interpréter la valeur extraite en complément à deux, commencez par décaler la plage à extraire de manière à ce que son bit de poids fort se trouve à l'index 31, puis faites un second décalage pour que son bit de poids faible se trouve à l'index 0.)

On suppose que $0 <$ `size` < 32, $0 \leq$ `start` < 32 et $0 <$ `start` $+$ `size` ≤ 32.

Réponse :

Partie 3 [20 points] Écrivez le corps de la méthode `decompress`, qui extrait les 1, 2 ou 3 différences contenues dans les 16 bits de poids faible de `value`, en les interprétant comme spécifié dans la table plus haut. Ces différences sont utilisées pour calculer et stocker la valeur du même nombre d'échantillons, ainsi : la première différence est ajoutée à la valeur de `samples[index]` pour obtenir le premier échantillon, qui est stocké dans `samples[index+1]` ; l'éventuelle seconde différence est ajoutée à ce premier échantillon pour obtenir le second, stocké dans `samples[index+2]` ; et l'éventuelle troisième différence est ajoutée au second échantillon pour obtenir le troisième, stocké dans `samples[index+3]`. Finalement, le nombre d'échantillons stockés est retourné. On suppose que `index` $+ 3 <$ `samples.length`.

L'extrait de test JUnit suivant, qui doit s'exécuter avec succès, illustre le fonctionnement de cette méthode :

```
short[] samples = {1096, 1072, 0, 0, 0};
int value = 0b00_0000011_1111111; // [+3, -1]
assertEquals(2, decompress(value, samples, 1));
assertArrayEquals(new short[]{1096, 1072, 1075, 1074, 0}, samples);
```

Réponse :

Réponse (suite) :

5 Valeurs optionnelles [25 points]

Un problème récurrent en programmation est celui de la représentation des valeurs optionnelles, c.-à-d. des valeurs potentiellement inexistantes.

Par exemple, la méthode `get` de l'interface `Map` retourne la valeur associée à la clef qu'on lui passe, et il n'est pas évident de savoir ce qu'elle doit retourner lorsque la clef ne se trouve pas dans la table. La solution retenue par les concepteurs de cette interface est de retourner **`null`** dans ce cas.

Même si cette solution fonctionne, et est couramment utilisée en Java, elle possède certains problèmes : d'une part, le fait que `get` peut retourner **`null`** n'est pas apparent dans son type de retour ; d'autre part, les appelants doivent vérifier systématiquement que la valeur retournée n'est pas nulle avant de l'utiliser, et il est facile d'oublier de le faire.

Une meilleure solution consiste à utiliser une classe pour représenter les valeurs optionnelles. Cette classe est similaire à une collection qui est soit vide, soit contient un seul élément. La bibliothèque Java offre depuis peu une telle classe, nommée `Optional`, et le but de cet exercice est d'en écrire une version simplifiée, nommée `Opt`.

L'utilisation de cette classe est illustrée ci-dessous. Les deux extraits de programmes sont équivalents et obtiennent de la table associative `table`, de type `Map<String,Double>`, la valeur associée à la clef `a`, avant de stocker dans `s` la chaîne qui est soit la représentation textuelle de cette valeur, soit la chaîne `???` si la clef ne figure pas dans la table. La version de gauche est « classique » et se base sur la méthode `get` qui retourne **`null`** lorsque la clef ne figure pas dans la table ; celle de droite suppose que la table possède une méthode `getOpt` qui retourne une valeur optionnelle de type `Opt<Integer>`, qui est vide lorsque la clef ne figure pas dans la table.

```
Double w = table.get("a");
String s = w != null
  ? String.valueOf(w)
  : "???";
```

```
String s = table.getOpt("a")
  .map(String::valueOf)
  .orElse("???");
```

Partie 1 [5 points] Écrivez la définition de la classe instanciable `Opt` et dotez-la d'un attribut final destiné à contenir la valeur optionnelle, ainsi que d'un constructeur privé prenant cette valeur en argument. L'attribut contiendra la valeur en question si elle existe, ou **`null`** sinon.

Bien entendu, cet attribut pouvant être d'un type quelconque, la classe doit être générique, et son paramètre de type représenter le type de la valeur optionnelle.

Réponse :

Partie 2 [5 points] Ajoutez à votre classe les deux méthodes de construction suivantes, qui doivent être génériques et statiques :

1. `empty`, qui ne prend aucun argument et retourne une valeur optionnelle vide,
2. `of`, qui prend en argument une valeur d'un type quelconque et retourne une instance de `Opt` contenant cette valeur, ou lève `NullPointerException` si cette valeur est **`null`**.

Réponse :

Partie 3 [5 points] Ajoutez à votre classe une méthode nommée `orElse`, qui prend en argument une valeur — éventuellement nulle — du même type que celle contenue dans le récepteur (**`this`**), et retourne la valeur contenue dans le récepteur s'il n'est pas vide, ou la valeur donnée en argument sinon.

Réponse :

Partie 4 [5 points] Ajoutez à votre classe une méthode nommée `ifPresent`, qui prend en argument un consommateur et appelle sa méthode `accept` avec la valeur contenue dans le récepteur si — et seulement si — celui-ci n'est pas vide.

Pour mémoire, un consommateur est une instance d'une classe implémentant l'interface fonctionnelle `Consumer` suivante :

```
public interface Consumer<T> { void accept(T t); }
```

Réponse :

Partie 5 [5 points] Ajoutez à votre classe une méthode générique nommée `map` prenant en argument une fonction et retournant la valeur optionnelle résultant de l'application de cette fonction à la valeur contenue dans le récepteur, ou une valeur optionnelle vide si ce dernier l'est également. Le type de retour de la fonction peut être quelconque, raison pour laquelle `map` doit être générique.

Pour mémoire, une fonction est une instance d'une classe implémentant l'interface fonctionnelle `Function` suivante :

```
public interface Function<T, R> { R apply(T t); }
```

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 ou descriptions parfois simplifiés.

Classe `FileReader`

La classe `java.io.FileReader`, qui hérite de `Reader`, représente un flot d'entrée de caractères qui obtient ses données d'un fichier.

```
public class FileReader {
  // Construit un flot d'entrée de caractères qui obtient ses données du fichier nommé
  // fileName, et dont le contenu est encodé en UTF-8.
  public FileReader(String fileName);
}
```

Classe `BufferedReader`

La classe `java.io.BufferedReader`, qui hérite de `Reader`, offre la possibilité de lire les données d'un flot de caractères ligne par ligne.

```
public class BufferedReader {
  // Construit une instance de BufferedReader qui obtient ses caractères du flot in,
  // que l'on nomme le flot sous-jacent.
  public BufferedReader(Reader in);

  // Lit et retourne la prochaine ligne du flot sous-jacent, ou null s'il est terminé.
  public String readLine() throws IOException;

  // Ignore les n prochains caractères du flot sous-jacent.
  public void skip(long n);
}
```

(Suite à la page suivante)

Classe `Arrays`

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

```
public class Arrays {
  // Retourne une copie du tableau array de longeur newLength.
  // Les éventuels nouveaux éléments valent 0.
  static byte[] copyOf(byte[] array, int newLength);

  // Trie les éléments du tableau array par ordre croissant.
  static void sort(long[] array);

  // Retourne l'index de l'élément key dans le tableau array, qui doit être trié en ordre
  // croissant. Si key ne se trouve pas dans array, la valeur -i-1 est retournée, où i est
  // la position à laquelle key devrait être insérée dans le tableau pour qu'il reste trié
  // par ordre croissant. (Note : 0 ≤ i ≤ array.length)
  static int binarySearch(long[] array, long key);

  // Retourne vrai ssi les tableaux array1 et array2 contiennent les mêmes éléments.
  static boolean equals(byte[] array1, byte[] array2);

  // Retourne une valeur de hachage basée sur le contenu du tableau array.
  static int hashCode(byte[] array);
}
```

Classe `Integer`

La classe `java.lang.Integer` contient entre autres des méthodes statiques travaillant sur les entiers de type `int`.

```
public class Integer {
  // Convertit l'entier i en une valeur de type long, en l'interprétant comme non signé.
  static long toUnsignedLong(int i);

  // Retourne l'entier dont la chaîne s est la représentation textuelle en base b,
  // ou lève NumberFormatException si s ne représente pas un entier valide.
  static int parseInt(String s, int b) throws NumberFormatException;
}
```

Classe `String`

La classe `java.lang.String`, immuable, représente une chaîne de caractères.

```
public class String {
  // Retourne la sous-chaîne allant du caractère d'index b (inclus) à celui d'index e (exclu).
  String substring(int b, int e);
}
```

(Suite à la page suivante)

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 le nombre de paires clef/valeur présentes dans la table.
  int size();

  // Retourne vrai ssi la table contient la clef k.
  boolean containsKey(K k);

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

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

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

  // Retourne une vue sur l'ensemble des paires clef/valeur.
  Set<Map.Entry<K, V>> entrySet();
}
```

Interface `Map.Entry`

L'interface `java.util.Map.Entry` représente une paire clef/valeur.

```
public interface Map.Entry<K, V> {
  // Retourne la clef de la paire.
  K getKey();

  // Retourne la valeur de la paire.
  V getValue();
}
```