Pratique de la programmation orientée-objet
Examen intermédiaire

3 avril 2019

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 Dates empaquetées [23 points]

Dans le calendrier grégorien en usage Suisse et dans la plupart du monde occidental aujourd'hui, une date est constituée de trois composantes entières : l'année, le mois et le jour. Pour cet exercice, nous ferons l'hypothèse qu'une date est valide si :

1. l'année est comprise entre 1 et 3000 (inclus),
2. le mois est compris entre 1 (janvier) et 12 (décembre), inclus,
3. le jour est compris entre 1 et le nombre de jour dans le mois, inclus.

Le nombre de jours dans un mois varie et vaut :

1. 31 pour janvier (1), mars (3), mai (5), juillet (7), août (8), octobre (10) et décembre (12),
2. 30 pour avril (4), juin (6), septembre (9) et novembre (11),
3. 28 pour février (2) si l'année n'est pas bissextile, 29 sinon.

Finalement, une année est bissextile si elle est divisible par 4 mais pas par 100, ou si elle est divisible par 400. Par exemple, 1900 n'est pas bissextile, mais 2000 l'est.

Une date peut être empaquetée dans un entier de 32 bits de la manière suivante :

Bits	31 à 21	20 à 9	8 à 5	4 à 0
Contenu	0	année	mois	jour

La classe `PackedDate` ci-dessous, à compléter, fournit des méthodes statiques permettant de manipuler des dates empaquetées ainsi dans des entiers de type `int`.

```
public final class PackedDate {
  // Index de premier bit et taille en bits des composantes
  private static final int DAY_START   = 0, DAY_SIZE   =  5;
  private static final int MONTH_START = 5, MONTH_SIZE =  4;
  private static final int YEAR_START  = 9, YEAR_SIZE  = 12;

  private PackedDate() {}

  public static boolean isLeapYear(int y) { à faire }
  public static int daysInMonth(int y, int m) { à faire }

  public static int pack(int y, int m, int d) { à faire }

  public static int year(int pkDate) { … }
  public static int month(int pkDate) { à faire }
  public static int day(int pkDate) { … }

  public static String toString(int pkDate) { à faire }
}
```

Partie 1 [4 points] Écrivez le corps de la méthode `isLeapYear`, qui retourne vrai ssi l'année `y` est bissextile, ou lève `IllegalArgumentException` si elle n'est pas dans les limites données plus haut.

Réponse :

Partie 2 [5 points] Écrivez le corps de la méthode `daysInMonth`, qui retourne la durée en jours du mois `m` de l'année `y`, ou lève `IllegalArgumentException` si l'une de ces valeurs n'est pas dans les limites données plus haut.

Réponse :

Partie 3 [4 points] Écrivez le corps de la méthode `pack`, qui empaquète l'année `y`, le mois `m` et le jour `d` ou lève `IllegalArgumentException` si l'une de ces valeurs n'est pas dans les limites données plus haut.

Réponse :

Partie 4 [3 points] Écrivez le corps de la méthode `month`, qui retourne le mois de la date empaquetée `pkDate`, supposée valide.

Réponse :

Partie 5 [5 points] Écrivez le corps de la méthode `toString`, qui retourne la représentation textuelle de la date empaquetée `pkDate`, supposée valide.

Cette représentation textuelle consiste en l'année, suivie du mois puis du jour. Ces trois composantes sont séparées par un tiret (`-`) et le mois et le jour font toujours exactement 2 caractères, le premier étant `0` au besoin. Par exemple, la représentation textuelle du 3 avril 2019 est `2019-04-03`.

Vous pouvez faire l'hypothèse que toutes les méthodes de la classe `PackedDate`, y compris `year` et `day` que vous n'avez pas dû écrire mais qui retournent respectivement l'année et le jour de la date empaquetée donnée, existent et fonctionnent correctement.

Réponse :

Partie 6 [2 points] La définition de la méthode `toString` ci-dessus n'inclut pas l'annotation `@Override`. D'après vous, devrait-elle être ajoutée ou non ? Justifiez votre réponse.

Réponse :

2 Dates [14 points]

Au moyen de la classe `PackedDate` de l'exercice précédent, il est facile d'écrire une classe `Date` immuable représentant une date sous la forme d'un objet.

Le but de cet exercice est d'esquisser la définition d'une telle classe en écrivant certaines de ses méthodes principales. Pour cela, vous pouvez faire l'hypothèse que toutes les méthode de la classe `PackedDate` existent, et fonctionnent correctement.

Partie 1 [5 points] Écrivez le début de la définition d'une classe immuable nommée `Date` représentant une date, stockée en interne sous forme empaquetée. Pour cette première partie, votre définition doit consister en :

1. la classe elle-même,
2. son ou ses attribut(s),
3. une méthode statique nommée `of`, utilisable comme illustré ci-dessous,
4. un constructeur privé.

La méthode `of` prend en argument une année, un mois et un jour et retourne la date correspondante, ou lève `IllegalArgumentException` si l'un de ces arguments est invalide (cette tâche peut être laissée à `PackedDate`). Par exemple, l'objet `Date` correspondant au 3 avril 2019 peut être créé ainsi :

```
Date d = Date.of(2019, 4, 3);
```

Réponse :

Partie 2 [4 points] Pensez-vous que la classe `Date` doit redéfinir les méthodes `hashCode` et `equals` de `Object` ? Justifiez votre réponse, et si elle est affirmative, écrivez le code des deux redéfinitions.

Réponse :

Partie 3 [5 points] Les dates étant ordonnées, il est sensé que la classe `Date` implémente l'interface `Comparable` de la bibliothèque Java.

Montrez comment la définition de la classe devrait être modifiée pour qu'elle implémente cette interface, et écrivez la définition de la méthode `compareTo` correspondante, de manière à ce qu'une date antérieure à une autre soit considérée comme plus petite qu'elle.

Par exemple, la valeur affichée par l'extrait de programme suivant devrait être négative :

```
Date d1 = Date.of(2019, 2, 18);
Date d2 = Date.of(2019, 4,  3);
System.out.println(d1.compareTo(d2));
```

Réponse :

3 Ensemble d'énumération [23 points]

En plus des classes `HashSet` et `TreeSet` décrites dans le cours, la bibliothèque Java offre une autre mise en œuvre des ensembles nommée `EnumSet`. Cette mise en œuvre est très efficace, mais ne peut être utilisée que lorsque le type des éléments de l'ensemble est un type énuméré (`enum`).

La manière dont `EnumSet` fonctionne est simple à comprendre si on fait l'hypothèse que tous les types énumérés ont au plus 64 valeurs. Dans ce cas, le contenu de l'ensemble peut simplement être représenté au moyen d'un entier de type `long`, dans lequel le bit n correspond au n^{e} élément de l'énumération. Ce bit vaut 1 si l'élément appartient à l'ensemble, 0 sinon. Pour les types énumérés ayant plus de 64 éléments, `EnumSet` utilise une représentation plus complexe, mais nous nous limiterons ici aux types énumérés ayant au plus 64 éléments.

Le but de cet exercice est de compléter une version simplifiée de la classe `EnumSet`, qui se présente ainsi :

```
public final class EnumSet<E extends Enum<E>> {
  private final E[] values;
  private long set;

  private EnumSet(E[] values, long initialSet) {
    if (! (values.length <= Long.SIZE))
      throw new IllegalArgumentException();
    this.values = Arrays.copyOf(values, values.length);
    this.set = initialSet;
  }

  public static <E extends Enum<E>> EnumSet<E> noneOf(E[] values)
    { à faire }

  public static <E extends Enum<E>> EnumSet<E> range(E[] values,
                                                     E from,
                                                     E to)
    { à faire }

  public boolean isEmpty() { à faire }
  public int size() { à faire }

  public boolean contains(E e) { à faire }
  public boolean containsAll(EnumSet<E> that) { à faire }

  public boolean add(E e) { à faire }
  public boolean addAll(EnumSet<E> that) { à faire }

  @Override
  public String toString() { à faire }
}
```

Notez que le paramètre de type `E` de la classe et des méthodes statiques est borné par `Enum<E>`, ce qui garantit qu'il s'agit d'un type énuméré. Le type `Enum` est résumé en annexe.

L'attribut `values`, passé au constructeur, est destiné à contenir la totalité des valeurs de l'énumération, dans l'ordre. Autrement dit, il doit contenir un tableau identique à celui retourné par la méthode statique `values` du type énuméré des éléments de l'ensemble.

Partie 1 [2 points] Écrivez le corps de la méthode `noneOf`, qui retourne un nouvel ensemble vide pour le type énuméré dont les valeurs sont données en argument, ou lève `IllegalArgumentException` si le nombre de valeurs est supérieur à 64. Cette vérification étant déjà faite par le constructeur donné plus haut, elle n'a pas besoin d'être refaite dans `noneOf`.

Notez bien que la méthode `noneOf` a pour but d'être appelée avec le résultat de la méthode `values` du type énuméré en argument, malgré le fait qu'elle crée un ensemble vide ! L'exemple ci-dessous illustre son utilisation correcte pour créer un ensemble vide de jours :

```
enum WeekDay { MON, TUE, WED, THU, FRI, SAT, SUN }
EnumSet<WeekDay> noDays = EnumSet.noneOf(WeekDay.values());
```

Réponse :

Partie 2 [5 points] Écrivez le corps de la méthode `range`, qui crée un ensemble contenant toutes les valeurs du type énuméré comprises dans l'intervalle défini par les valeurs `from` et `to`, toutes deux incluses. Lève `IllegalArgumentException` si le tableau de valeurs contient plus de 64 valeurs, ou si `to` apparaît avant `from` dans l'énumération. Attention, votre mise en œuvre ne doit *pas* contenir de boucle.

Tout comme pour `noneOf`, le premier argument doit contenir toutes les valeurs du type énuméré, telles que retournées par la méthode `values`. L'exemple suivant illustre l'utilisation correcte de `range` pour créer un ensemble des jours travaillés, étant donné le type énuméré `WeekDay` ci-dessus :

```
EnumSet<WeekDay> workingDays =
  EnumSet.range(WeekDay.values(), WeekDay.MON, WeekDay.FRI);
```

Réponse :

(suite à la page suivante)

Réponse (suite) :

Partie 3 [4 points] Écrivez le corps des méthodes `isEmpty` et `size`. La première retourne vrai ssi l'ensemble est vide, tandis que la seconde retourne le nombre d'éléments qu'il contient.

Votre mise en œuvre de `isEmpty` ne doit *pas* appeler la méthode `size`.

Réponse :

Partie 4 [4 points] Écrivez le corps des méthodes `contains` et `containsAll`. La première retourne vrai ssi l'ensemble contient l'élément donné, tandis que la seconde retourne vrai ssi l'ensemble contient la totalité des éléments de l'ensemble donné.

Votre mise en œuvre de `containsAll` ne doit *pas* contenir de boucle.

Réponse :

Partie 5 [4 points] Écrivez le corps des méthodes `add` et `addAll`. La première ajoute l'élément donné à l'ensemble, tandis que la seconde y ajoute la totalité des éléments de l'ensemble donné. Les deux retournent vrai ssi le contenu de l'ensemble a changé suite à l'ajout.

Votre mise en œuvre de `addAll` ne doit *pas* contenir de boucle.

Réponse :

Partie 6 [4 points] Écrivez le corps de la méthode `toString`, qui retourne la représentation textuelle de l'ensemble. Celle-ci consiste en la représentation textuelle des éléments, dans l'ordre, séparées par des virgules et entourées d'accolades.

Par exemple, pour l'ensemble `workingDays` défini dans la partie 2, la méthode `toString` doit retourner la chaîne `{MON,TUE,WED,THU,FRI}`.

Réponse :

4 Dictionnaire pour mots croisés [15 points]

Un dictionnaire pour mots croisés est un dictionnaire conçu pour faciliter la résolution de mots croisés. Un exemple simple d'un tel dictionnaire en est un dans lequel les mots sont triés d'abord par longueur, puis par ordre alphabétique.

Le but de cet exercice est d'écrire une classe nommée `CrosswordDict` représentant un tel dictionnaire, offrant trois méthodes :

1. `add`, qui prend un mot de type `String` en argument et l'ajoute au dictionnaire s'il ne s'y trouve pas déjà, ou lève `IllegalArgumentException` si le mot est vide,

2. `addAll`, qui prend une collection de mots de type `Collection<String>` en argument et ajoute au dictionnaire la totalité des mots qu'elle contient et qui ne s'y trouvent pas déjà, ou lève `IllegalArgumentException` si l'un des mots est vide ; notez que la collection elle-même a le droit d'être vide,

3. `print`, qui imprime à l'écran le contenu du dictionnaire, selon le format donné plus bas.

L'extrait de programme suivant illustre l'utilisation de cette classe :

```
CrosswordDict d = new CrosswordDict();
d.add(”arbre”);
d.add(”maison”);
d.addAll(Arrays.asList(”chaise”,
                       ”table”,
                       ”bois”,
                       ”bureau”,
                       ”arbre”,
                       ”crayon”));
d.print();
```

En l'exécutant, il devrait produire la sortie suivante :

```
Mots de 4 lettres :
  bois
Mots de 5 lettres :
  arbre
  table
Mots de 6 lettres :
  bureau
  chaise
  crayon
  maison
```

dans laquelle les mots apparaissent groupés par longueur, les groupes sont triés par longueur croissante, et les mots de chaque groupe sont triés dans l'ordre alphabétique.

(réponse à la page suivante)

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 `Map`

L'interface `java.util.Map` représente les tables associatives. Elle est implémentée, entre autres, par les classes `HashMap` et `TreeMap`.

```
public interface Map<K, V> {
  // Associe la valeur value à la clef key. Retourne la valeur qui était associée
  // à la clef, ou null s'il n'y en avait pas.
  V put(K key, V value);

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

  // Retourne vrai si et seulement si la table contient la clef key.
  boolean containsKey(K key);

  // Retourne 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();
}
```

Interface `Set`

L'interface `java.util.Set` représente un ensemble de valeurs. Elle est implémentée, entre autres, par les classes `HashSet` et `TreeSet`.

```
public interface Set<E> {
  // Ajoute l'élément e à l'ensemble et retourne vrai ssi il a été modifié
  // en conséquence.
  boolean add(E e);
}
```

Interface `Comparable`

L'interface `Comparable` est destinée à être implémentée par toutes les classes dont les instances peuvent être comparées entre elles.

```
public interface Comparable<T> {
  // Compare this et that et retourne un entier négatif si this < that,
  // zéro si this == that, et un entier positif sinon.
  int compareTo(T that);
}
```

Classe `String`

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

```
public class String {
  // Retourne la représentation textuelle en base 10 de l'entier i.
  static String valueOf(int i);

  // Retourne la longueur de la chaîne à laquelle on l'applique.
  int length();
}
```

Classe `StringJoiner`

La classe `java.lang.StringJoiner` représente un « joigneur » de chaîne, qui permet de construire des chaînes de caractères constituées d'une séquence de chaînes séparées par un délimiteur et entourée 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();
}
```

(suite à la page suivante)

Classe `Enum`

La classe `java.lang.Enum` sert de classe mère à toutes les énumérations.

```
public class Enum<E extends Enum<E>> {
  // Retourne la position du récepteur dans l'énumération à laquelle il appartient.
  int ordinal();
}
```

Classe `Integer`

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

```
public class Integer {
  // Compare x et y et retourne un entier négatif si x < y, zéro si x == y,
  // et un entier positif sinon.
  static int compare(int x, int y);
}
```

Classe `Long`

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

```
public class Long {
  // Retourne le nombre de bits valant 1 dans l'entier l.
  static int bitCount(long l);
}
```