# Pratique de la programmation orientée-objet
# Examen intermédiaire

11 avril 2018

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 Conversion d'entiers [22 points]

La classe `IntConverter` ci-dessous, à compléter, contient des méthodes de conversion entre des entiers non négatifs (c-à-d positifs ou nuls) et leur représentation en base 10.

```
public final class IntConverter {
  public static String posIntToStr(int i) { à faire }
  public static int strToPosInt(String s) { à faire }
}
```

**Partie 1 [8 points]** Écrivez le corps de la méthode `posIntToStr`, qui convertit l'entier positif ou nul qu'elle reçoit en la plus courte (c-à-d sans zéros inutiles en tête) chaîne de caractères le représentant en base 10, ou alors lève l'exception `IllegalArgumentException` si cet entier est négatif. Les deux assertions JUnit ci-dessous, qui doivent s'exécuter avec succès, illustrent son utilisation :

```
assertEquals("2018", IntConverter.posIntToStr(2018));
assertThrows(IllegalArgumentException.class,
             () -> { IntConverter.posIntToStr(-1); });
```

Notez que vous n'avez **pas** le droit d'utiliser des méthodes de la bibliothèque Java qui font la conversion pour vous, même de manière implicite, car le but de cet exercice est de la faire vous-même. Dès lors, votre méthode doit produire les caractères de la chaîne les uns après les autres, et chaque caractère doit être obtenu **au moyen d'opérations arithmétiques uniquement**.

Souvenez-vous qu'en Java, les caractères sont des entiers. De plus, les caractères correspondants aux chiffres décimaux sont encodés par des entiers successifs commençant à 48 et ordonnés par poids. Ainsi, le caractère `'0'` est encodé par 48, le caractère `'1'` par 49, et ainsi de suite jusqu'au caractère `'9'`, encodé par 57.

En conséquence, la conversion d'un entier compris entre 0 et 9 (inclus) en le caractère le représentant peut se faire au moyen d'une simple addition, comme l'illustre l'exemple suivant :

```
char c = '0' + 3;                // c == '3'
```

Réponse :

*(suite à la page suivante)*

Réponse (suite) :

**Partie 2 [2 points]** La méthode `posIntToStr` que vous venez d'écrire ne fonctionne que pour les entiers positifs. Une méthode plus générale, gérant également les entiers négatifs, pourrait être écrite ainsi :

```
public static String intToStr(int i) {
  return i < 0 ? "-" + posIntToStr(-i) : posIntToStr(i);
}
```

Cette mise en œuvre est-elle correcte ? Justifiez votre réponse.

Réponse :

**Partie 3 [8 points]** Écrivez le corps de la méthode `strToPosInt`, qui convertit la chaîne qu'elle reçoit en l'entier qu'elle représente en base 10. Votre méthode doit lever `IllegalArgumentException` si la chaîne ne représente pas un entier valide, et `NullPointerException` si elle vaut `null`. Les assertions JUnit ci-dessous, qui devraient s'exécuter avec succès, illustrent son utilisation :

```
assertEquals(2018, IntConverter.strToPosInt("2018"));
assertEquals(2018, IntConverter.strToPosInt("002018"));
assertThrows(IllegalArgumentException.class,
             () -> { IntConverter.strToPosInt("12a"); });
assertThrows(IllegalArgumentException.class,
             () -> { IntConverter.strToPosInt(""); });
```

Vous pouvez faire l'hypothèse que l'entier représenté par la chaîne est inférieur ou égal à la plus grande valeur de type `int` (2 147 483 647).

Notez que, une fois encore, vous n'avez **pas** le droit d'utiliser des méthodes de la bibliothèque Java pour effectuer la conversion, et devez donc la faire caractère par caractère, **au moyen d'opérations arithmétiques uniquement**.

Réponse :

**Partie 4 [4 points]** Montrez ce que vous devriez changer dans votre méthode `strToPosInt` pour qu'elle produise un entier décimal codé binaire plutôt qu'un entier normal. Vous n'avez pas besoin de récrire la totalité du code, mais seulement les parties qui diffèrent de la version originale ci-dessus, et vous pouvez cette fois faire l'hypothèse que l'entier représenté par la chaîne est inférieur ou égal à 99 999 999.

Pour mémoire, dans un entier décimal codé binaire, chaque chiffre décimal est représenté au moyen d'exactement 4 bits. Par exemple, l'entier 2018 est représenté par les 16 bits suivants : `0010 0000 0001 1000`.

Réponse :

## 2 Comptage de bits [20 points]

Il est parfois utile de pouvoir compter le nombre de bits valant 1 dans un vecteur de bits, opération nommée *comptage de bits*. Le but de cet exercice est d'écrire une méthode de comptage de bits et de l'utiliser pour calculer la distance de Hamming (décrite plus bas) entre deux vecteurs de bits.

**Partie 1 [8 points]** Écrivez une première version de la méthode `bitCount` qui compte le nombre de bits valant 1 dans une valeur de type `int` (32 bits). Utilisez la technique évidente consistant à tester chaque bit individuellement dans une boucle.

Cette méthode doit être utilisable comme illustré par les assertions JUnit suivantes, qui devraient s'exécuter avec succès :

```
assertEquals( 0, bitCount(0));
assertEquals(32, bitCount(-1));
assertEquals( 8, bitCount(0b1111_1111));
```

Réponse :

**Partie 2 [2 points]** Une application du comptage de bits est le calcul de la **distance de Hamming** entre deux vecteurs de bits, qui est simplement le nombre de bits qui diffèrent entre les deux vecteurs. Par exemple, la distance de Hamming entre les vecteurs de bits `0010` et `0001` vaut 2, car deux bits (d'index 0 et 1) diffèrent.

Il s'avère très facile d'écrire une méthode calculant la distance de Hamming entre deux vecteurs de bits de type `int` au moyen de la méthode `bitCount` ci-dessus. Montrez cela en écrivant une méthode nommée `hammingDistance` utilisable comme illustré par les assertions JUnit suivantes, qui devraient s'exécuter avec succès :

```
assertEquals( 2, hammingDistance(0b0010, 0b0001));
assertEquals( 4, hammingDistance(0b0101, 0b1010));
assertEquals(32, hammingDistance(0, -1));
```

Réponse :

**Partie 3 [10 points]** Compter les bits au moyen d'une boucle, comme dans la partie 1, est inefficace. En utilisant judicieusement l'addition, il est possible de faire mieux, comme l'exemple suivant l'illustre.

Admettons que l'on désire compter les bits dans le vecteur de 4 bits `abcd`, où `a`, `b`, `c` et `d` sont les bits individuels. Il est facile de voir que le nombre de bits valant 1 est simplement la somme des bits individuels, c-à-d `a+b+c+d`. Il se trouve que cette somme peut se calculer efficacement, au moyen de *deux* additions seulement.

Pour comprendre cela, considéreons les deux vecteurs de 4 bits suivants, obtenus à partir du vecteur original :

1. `0b0d`, obtenu en mettant à 0 (au moyen d'un *et* bit à bit) les bits 1 et 3 du vecteur original,

2. `0a0c`, obtenu en décalant d'un bit vers la droite le vecteur original, puis en mettant à 0 (au moyen d'un *et* bit à bit) ses bits 1 et 3.

En additionnant ces deux vecteurs au moyen de l'addition 4 bits standard, on obtient un vecteur de 4 bits dont les deux bits de poids faible contiennent la somme `c+d`, et les deux bits de poids fort la somme `a+b`. Remarquez qu'au moyen d'une seule addition, nous avons calculé à la fois `a+b` *et* `c+d`.

Il est désormais facile de calculer le résultat final au moyen d'une seconde addition des deux bits de poids faible et des deux bits de poids fort de cette valeur. La totalité du processus est illustrée dans la figure 1.

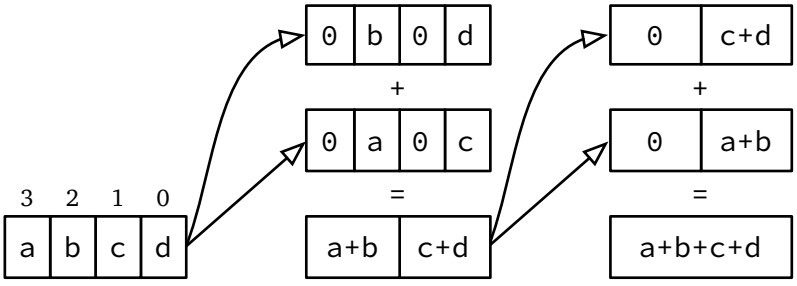


Fig. 1: Comptage de bits rapide

La même idée peut être utilisée pour compter le nombre de bits valant 1 dans un vecteur de 32 bits de type `int`, en utilisant seulement 5 additions ainsi que plusieurs *et* bit à bit et décalages.

Utilisez cette technique pour écrire une deuxième version, plus efficace, de la méthode `bitCount`. N'essayez pas d'utiliser une boucle pour effectuer les différentes opérations, il est plus simple (et plus rapide) de les effectuer individuellement.

Réponse :

## 3 Table associative d'énumération [33 points]

En plus des classes `HashMap` et `TreeMap` décrites dans le cours, la bibliothèque Java offre une autre mise en œuvre des tables associatives nommée `EnumMap`. Cette mise en œuvre est très efficace, mais ne peut être utilisée que lorsque le type des clefs de la table est un type énuméré (`enum`).

La manière dont `EnumMap` est mise en œuvre est simple : les valeurs de la table sont stockées dans un tableau, dont la taille est égale au nombre de clefs — c-à-d la taille de l'énumération des clefs. La valeur correspondant à une clef est simplement stockée à l'index de la clef dans l'énumération, qui peut être obtenu au moyen de la méthode `ordinal`. Si un élément du tableau est `null`, cela signifie qu'aucune valeur n'existe pour cette clef dans la table. En conséquence, il est interdit de stocker `null` comme valeur dans une telle table.

Le but de cet exercice est d'écrire une version simplifiée de la classe `EnumMap`. Au lieu de lui faire implémenter l'interface `Map` de la bibliothèque Java, nous lui ferons implémenter l'interface simplifiée suivante :

```
public interface SimpleMap<K, V> {
  int size();
  boolean containsKey(K k);
  V get(K key);
  V getOrDefault(K k, V d);
  void put(K k, V v);
  void putAll(SimpleMap<K, V> that);
  V remove(K k);
  void clear();
}
```

Toutes les méthodes de cette interface doivent lever `NullPointerException` si l'un de leurs arguments est `null`, à l'exception de `getOrDefault`, qui doit accepter `null` comme *second* argument.

La déclaration de la classe `EnumMap` et de son constructeur sont donnés ci-après. Notez que le paramètre de type `K` est borné par `Enum<K>`, ce qui garantit que les clefs appartiennent à une énumération. Le type `Enum` est décrit brièvement dans le formulaire en annexe.

```
public final class EnumMap<K extends Enum<K>, V>
    implements SimpleMap<K, V> {

  private final K[] allKeys;
  private final V[] values;
  private int size;

  public EnumMap(K[] allKeys) {
    this.allKeys = Arrays.copyOf(allKeys, allKeys.length);
    this.values = (V[]) new Object[allKeys.length];
    this.size = 0;
  }

  à faire : autres méthodes
}
```

L'attribut `values` a pour but de contenir les valeurs de la table. Comme expliqué ci-dessus, lorsqu'un élément de ce tableau vaut `null`, cela signifie qu'aucune valeur n'est associée à la clef correspondante dans la table.

Le constructeur a pour but d'être appelé avec un tableau contenant la totalité des clefs, obtenu au moyen de la méthode statique `values` de l'énumération. L'exemple ci-dessous illustre une utilisation correcte de la classe `EnumMap` :

```
enum WkDay { MO, TU, WE, TH, FR, SA, SU }

SimpleMap<WkDay, String> frDays =
  new EnumMap<>(WkDay.values());
frDays.put(WkDay.MO, "lundi");
frDays.put(WkDay.TU, "mardi");
String mondayInFr = frDays.get(WkDay.MO);
System.out.println("In␣French,␣Monday␣is␣" + mondayInFr);
```

**Partie 1 [5 points]** Écrivez le corps des méthodes `size` et `containsKey`. Notez que grâce à la borne du paramètre de type `K`, il est possible d'appeler n'importe quelle méthode de la classe `Enum`, en particulier `ordinal`, sur un objet de type `K`.

Réponse :

**Partie 2 [6 points]** Écrivez le corps des méthodes `get` et `getOrDefault`. Notez que `get` doit retourner `null` lorsqu'aucune valeur n'est associée à la clef donnée dans la table, tandis que `getOrDefault` doit retourner la valeur qu'elle a reçu en second argument (qui peut être `null`) dans ce cas.

Réponse :

**Partie 3 [8 points]** Écrivez le corps des méthodes `put` et `putAll`. Les deux doivent lever l'exception `NullPointerException` si l'on tente d'insérer une valeur `null` dans la table.

Réponse :

**Partie 4 [7 points]** Écrivez le corps des méthodes `remove` et `clear`. Notez que `remove` doit retourner la valeur associée précédemment à la clef, s'il y en avait une, et `null` sinon.

Réponse :

**Partie 5 [7 points]** Écrivez le corps de la méthode `toString` de `EnumMap`, qui redéfinit celle de `Object` et produit une chaîne dans laquelle toutes les paires clef/-valeur de la table apparaissent, entourées d'accolades (`{` et `}`) et séparées par des virgules (`,`). La clef et la valeur de chaque paire sont séparées par un égal (`=`).

Le comportement de cette méthode est illustré par l'assertion JUnit suivante, qui devrait s'exécuter avec succès, `frDays` étant la table construite plus haut :

```
assertEquals("{MO=lundi,TU=mardi}", frDays.toString());
```

Réponse :

*Page laissée intentionnellement vide.*

# 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 `String`

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

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

### Classe `StringBuilder`

La classe `java.lang.StringBuilder` représente un bâtisseur de chaîne, qui permet de construire petit à petit une chaîne de caractères.

```
class StringBuilder {
  // Crée un bâtisseur de chaîne vide.
  StringBuilder();

  // Crée un bâtisseur de chaîne initialisé avec la chaîne donnée.
  StringBuilder(String s);

  // Retourne la longueur actuelle de la chaîne en cours de construction.
  int length();

  // Ajoute le caractère donné à la chaîne en cours de construction et retourne
  // this.
  StringBuilder append(char c);

  // Ajoute la représentation textuelle de l'objet donné, obtenue avec toString,
  // à la chaîne en cours de construction et retourne this.
  StringBuilder append(Object o);

  // Inverse la chaîne en cours de construction et retourne this.
  StringBuilder reverse();

  // Retourne la chaîne en cours de construction.
  String toString();
}
```

### Classe `Enum`

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

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

### Classe `Objects`

La classe `java.util.Objects`, non instanciable, contient des méthodes utilitaires sur les objets.

```
class Objects {
  // Retourne son argument s'il n'est pas nul, lève NullPointerException
  // sinon.
  static <T> T requireNonNull(T o);
}
```

### Classe `Arrays`

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

```
class Arrays {
  // Remplit le tableau avec la valeur donnée.
  static void fill(Object[] a, Object v);
}
```