Examen intermédiaire

CS-108

2025-04-11

1 Bâtisseur de chaînes [30 points]

La classe `StringBuilder` de la bibliothèque Java représente un bâtisseur de chaînes de caractères. Le but de cet exercice est de compléter la définition de la classe `SStringBuilder` ci-dessous, qui en constitue une version simplifiée n'offrant qu'un petit sous-ensemble des méthodes.

Cette classe stocke les caractères de la chaîne en cours de construction dans un tableau de caractères dont la taille — que l'on nomme la **capacité** (*capacity*) du bâtisseur — est supérieure ou égale à celle de la chaîne. Ce tableau de caractères est stocké dans l'attribut `chars`, tandis que la taille de la chaîne en cours de construction l'est dans l'attribut `size`. Les caractères de la chaîne en cours de construction se trouvent dans les `size` premiers éléments du tableau `chars`, les éventuels autres éléments ayant une valeur quelconque.

```
public final class SStringBuilder {
  private char[] chars = new char[8];
  private int size = 0;

  public int capacity() { ... }
  public int size() { ... }
  public char charAt(int index) { ... }
  public SStringBuilder append(String s) { ... }
  public SStringBuilder removeIf(Predicate<Character> predicate) { ... }
  public String build() { ... }
}
```

Partie 1 [2 points] Écrivez le corps des méthodes `capacity` et `size`, qui retournent respectivement la capacité du bâtisseur et la taille de la chaîne en cours de construction.

Partie 2 [2 points] Écrivez le corps de la méthode `charAt` qui retourne le caractère à l'index donné, ou lève une `IndexOutOfBoundsException` si cet index est invalide.

Partie 3 [12 points] Écrivez le corps de la méthode `append` qui ajoute la chaîne donnée à la fin de la chaîne en cours de construction, puis retourne le bâtisseur afin de permettre le chaînage des appels.

Votre mise en œuvre ne doit créer un nouveau tableau de caractères que si la taille de l'actuel ne permet pas de stocker la chaîne en cours de construction. La taille de ce nouveau tableau doit être la plus petite puissance de deux supérieure ou égale à la taille de la chaîne en cours de construction.

Pour copier les caractères de la chaîne à ajouter dans le tableau du bâtisseur, utilisez la méthode `getChars` de `String` décrite dans le formulaire en annexe.

Partie 4 [12 points] Écrivez le corps de la méthode `removeIf` qui supprime de la chaîne en cours de construction tous les caractères pour lesquels la méthode `test` du prédicat passé en argument retourne vrai, puis retourne le bâtisseur. L'interface fonctionnelle `Predicate` est définie ainsi :

```
public interface Predicate<T> { boolean test(T t); }
```

L'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre l'utilisation de cette méthode pour supprimer les voyelles (minuscules) de la chaîne en cours de construction :

```
String s = new SStringBuilder()
  .append("anticonstitutionnellement")
  .removeIf(c -> "aeiou".indexOf(c) != -1)
  .build();
assertEquals("ntcnstttnnllmnt", s);
```

Votre mise en œuvre ne doit traverser qu'une seule fois le tableau `chars`, qui doit être modifié directement. En d'autres termes, vous n'avez pas le droit de créer un nouveau tableau, ou d'utiliser un tableau ou une collection auxiliaire.

Partie 5 [2 points] Écrivez le corps de la méthode `build` qui retourne la chaîne en cours de construction. Aidez-vous pour cela du constructeur de `String` décrit dans le formulaire en annexe.

2 Tableau d'octets [28 points]

L'interface `ByteArray` ci-dessous représente un tableau d'octets non modifiable dont il est possible d'extraire non seulement des octets individuels (avec `getByte`), mais aussi des entiers de type `short` (avec `getShort`) en combinant deux octets consécutifs ; ces deux méthodes lèvent une `IndexOutOfBoundsException` si l'index donné est invalide. La méthode `size` retourne le nombre d'octets que contient le tableau, tandis que la méthode `slice` retourne une **tranche** du tableau, c.-à-d. une vue sur les éléments se trouvant entre les index donnés.

```
public interface ByteArray {
  int size();

  byte getByte(int index);
  short getShort(int index);

  default ByteArray slice(int fromInclusive, int toExclusive) { ... }
}
```

Le but de cet exercice est d'écrire deux classes implémentant cette interface, l'une permettant de voir un tableau de type `byte[]` comme une valeur de type `ByteArray`, et l'autre représentant une tranche d'un tableau existant.

Partie 1 [10 points] Écrivez une classe nommée `ByteArrayMSBF` implémentant `ByteArray` et permettant de voir un tableau d'octets de type `byte[]` comme une valeur de type `ByteArray`.

Le constructeur de `ByteArrayMSBF` prend en argument le tableau de type `byte[]` à voir comme une valeur de type `ByteArray` et le stocke **sans** le copier, car `ByteArrayMSBF` est une vue non modifiable, mais n'est pas immuable.

La méthode `getByte` retourne l'octet du tableau à l'index donné, tandis que `getShort` retourne l'entier `short` dont les 8 bits de poids fort sont ceux de l'octet à l'index donné et les 8 bits de poids faible sont ceux de l'octet suivant. Cette manière d'organiser les octets d'une valeur de

plus de 8 bits en commençant par l'octet de poids le plus fort est appelée *most significant byte first* (ou *big-endian*), d'où le nom de la classe.

Notez que comme la méthode `slice` de `ByteArray` est une méthode par défaut, vous n'avez pas besoin de la redéfinir.

L'utilisation de la classe `ByteArrayMSBF` est illustrée par l'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès :

```
byte[] a = {(byte) 0x12, (byte) 0x34, (byte) 0x56, (byte) 0x78};
ByteArray byteArray = new ByteArrayMSBF(a);

assertEquals(4, byteArray.size());
assertEquals(0x34, byteArray.getByte(1));
assertEquals(0x5678, byteArray.getShort(2));
```

Partie 2 [10 points] Écrivez une classe nommée `ByteArraySlice` implémentant `ByteArray` et représentant une « tranche » d'un tableau existant. Une tranche d'un tableau est simplement un de ses sous-tableaux, similaire à une sous-liste d'une liste.

Le constructeur de cette classe prend en arguments le tableau dont on désire extraire une tranche (de type `ByteArray`), l'index (inclusif) de début de la tranche et l'index (exclusif) de fin de la tranche. Il lève une `IndexOutOfBoundsException` si l'un des deux index est invalide, ou si l'index de fin est strictement inférieur à celui de début.

Notez que comme la méthode `slice` de `ByteArray` est une méthode par défaut, vous n'avez pas besoin de la redéfinir.

L'utilisation de la classe `ByteArraySlice` est illustrée par l'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès :

```
byte[] a = {(byte) 0x12, (byte) 0x34, (byte) 0x56, (byte) 0x78};
ByteArray byteArray = new ByteArraySlice(new ByteArrayMSBF(a), 1, 3);

assertEquals(2, byteArray.size());
assertEquals(0x34, byteArray.getByte(0));
assertEquals(0x3456, byteArray.getShort(0));
```

Partie 3 [2 points] Écrivez le corps de la méthode `slice` de l'interface `ByteArray`, qui retourne une tranche du tableau auquel on l'applique, entre les index de début et de fin donnés. Cette méthode lève les mêmes exceptions que le constructeur de `ByteArraySlice`, dans les mêmes circonstances.

Partie 4 [6 points] La méthode `slice` étant une méthode par défaut de `ByteArray`, elle est automatiquement héritée par les deux classes que vous avez écrites. Pensez-vous toutefois que, pour des raisons d'efficacité, il puisse valoir la peine de redéfinir cette méthode dans l'une ou l'autre (ou les deux) des classes ?

Si oui, dites pour laquelle ou lesquelles des classes une redéfinition serait plus efficace que la méthode par défaut, et écrivez son code. Si non, expliquez brièvement (en une phrase) pourquoi aucune redéfinition n'est nécessaire.

Suite à la page suivante

3 Détecteur de fins de lignes [17 points]

Les fins de lignes dans les fichiers textuels sont représentées au moyen de caractères de contrôle. Pour des raisons historiques, trois conventions existent pour représenter une fin de ligne :

1. au moyen d'un seul caractère CR (*carriage return*), noté `\r` en Java,
2. au moyen d'un seul caractère LF (*line feed*), noté `\n` en Java,
3. au moyen d'un caractère CR suivi d'un caractère LF.

Comme ces trois représentations existent, et que les deux dernières en tout cas se rencontrent fréquemment en pratique, il peut être utile d'avoir un moyen de déterminer la ou les représentation(s) utilisée(s) pour un fichier donné. C'est le but de la classe non instanciable `LineEndDetector` ci-dessous, à compléter dans le cadre de cet exercice. Le type énuméré `LineEnd` représente les trois représentations utilisées.

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

  public enum LineEnd {CR, LF, CR_LF}

  public static Set<LineEnd> lineEnds(Reader reader)
    throws IOException { ... }
  public static Set<LineEnd> lineEnds(String fileName)
    throws IOException { ... }
}
```

Partie 1 [12 points] Écrivez le corps de la première variante de la méthode `lineEnds`, qui prend un flot d'entrée de caractères en argument, et qui retourne l'ensemble des représentations de fin de ligne utilisées dans le flot.

Par exemple, si tous les caractères CR du flot sont suivis d'un caractère LF, et que le flot ne contient aucun autre caractère LF, alors l'ensemble retourné ne contient que l'élément `CR_LF`.

Votre méthode ne doit pas fermer le flot après avoir lu son contenu.

Partie 2 [5 points] Écrivez le corps de la seconde variante de la méthode `lineEnds`, qui prend un nom de fichier en argument et retourne l'ensemble des représentations de fin de ligne utilisées dans le fichier. On fait l'hypothèse que le fichier est encodé en UTF-8.

Votre méthode doit bien entendu se baser sur la première variante, et prendre garde à ce que le flot créé pour lire les caractères du fichier soit fermé dans tous les cas.

4 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 `Arrays`

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

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

Classe `Byte`

La classe `Byte` sert de classe d'emballage pour le type primitif `byte` et fournit aussi certaines méthodes statiques manipulant des valeurs de ce même type.

```
public class Byte {
  // Convertit l'octet x en entier de type int, en l'interprétant de manière non signée.
  static int toUnsignedInt(byte x);
}
```

Classe `String`

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

```
public final class String {
  // Construit une chaîne dont les caractères sont les count du tableau chars
  // à partir de l'index offset.
  public String(char[] chars, int offset, int count);

  // Copie les caractères de la chaîne dont l'index est compris entre srcFrom (inclusif)
  // et srcTo (exclusif) dans le tableau dst, à partir de l'index dstFrom (inclusif).
  public void getChars(int srcFrom, int srcTo, char[] dst, int dstFrom);

  // Retourne la longueur de la chaîne.
  public int length();
}
```

Interface `Set`

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

```
public interface Set<E> extends Iterable<E> {
  // Ajoute e à l'ensemble, et retourne vrai ssi son contenu a changé en conséquence.
  boolean add(E e);
}
```

Classe `Reader`

La classe abstraite `Reader` sert de classe mère à tous les flots d'entrée de caractères.

```
abstract public class Reader implements AutoCloseable {
  // Lit le prochain caractère du flot et le retourne sous la forme d'un entier de type int,
  // ou retourne -1 si la fin du flot a été atteinte.
  public abstract int read();

  // Ferme le flot et libère les éventuelles ressources associées.
  public abstract void close();
}
```

Classe `FileReader`

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

```
public class FileReader extends Reader {
  // 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);
}
```