Examen final

CS-108

2026-05-29

1 Union d'ensembles [25 points]

L'interface `Set` de la bibliothèque Java, qui représente un ensemble, n'offre malheureusement pas de méthode permettant de calculer l'union de deux ensembles sans modifier l'un d'eux. En effet, sa méthode `addAll` ajoute à l'ensemble auquel on l'applique — le récepteur — la totalité des éléments d'un autre ensemble, et détermine donc leur union, mais elle le fait en modifiant le récepteur.

Le but de cet exercice est de compléter la définition de la classe `UnionView` ci-dessous, qui permet de calculer l'union de deux ensembles sans devoir modifier l'un d'eux. Comme son nom l'indique, cette classe est une vue sur l'union de deux ensembles.

```
public final class UnionView<E> extends AbstractSet<E> {
  // à faire : attributs et constructeur

  @Override public int size() { /* à faire */ }
  @Override public Iterator<E> iterator() { /* à faire */ }
}
```

`UnionView` hérite de `AbstractSet`, une classe héritable de la bibliothèque Java dont le but est de faciliter la définition de classes représentant des ensembles. Elle implémente l'interface `Set` et contient une mise en œuvre de toutes les méthodes de cette interface sauf deux : `size` et `iterator`. Grâce à cela, `UnionView` peut se contenter de définir ces deux méthodes.

L'extrait de test JUnit suivant, qui doit s'exécuter avec succès, illustre le fonctionnement de la classe `UnionView` :

```
Set<String> s1 = Set.of("a", "d", "b");
Set<String> s2 = new HashSet<>(Set.of("c", "d", "b", "e"));
Set<String> s12 = new UnionView<>(s1, s2);

assertEquals(5, s12.size());
assertEquals(Set.of("a", "b", "c", "d", "e"), s12);

s2.clear();
assertEquals(s1, s12);
```

Partie 1 [3 points] Écrivez les attributs et le constructeur de la classe `UnionView` pour qu'elle puisse être utilisée comme dans l'extrait de test JUnit ci-dessus.

Notez bien que `UnionView` est une vue, et son constructeur ne doit donc pas calculer puis stocker l'union des deux ensembles reçus.

Partie 2 [6 points] Écrivez le corps de la méthode `size`, qui retourne le nombre d'éléments que contient l'union des deux ensembles.

Votre mise en œuvre ne doit en aucun cas créer une collection ou un tableau contenant les éléments de l'union pour déterminer sa taille. Elle ne doit pas non plus utiliser la méthode `iterator` définie dans la partie suivante.

Partie 3 [10 points] Écrivez le corps de la méthode `iterator`, qui retourne un itérateur permettant de parcourir les éléments de l'union, dans un ordre quelconque. La méthode `remove` de cet itérateur doit simplement lever une `UnsupportedOperationException`.

Votre mise en œuvre ne doit en aucun cas créer une collection ou un tableau contenant les éléments de l'union pour itérer sur eux. Elle peut par contre utiliser la méthode `size`.

Partie 4 [5 points] Même si les méthodes `size` et `iterator` sont les deux seules qui doivent absolument être définies dans `UnionView`, car elles sont abstraites dans `AbstractSet` pensez-vous qu'il pourrait également être judicieux de redéfinir la méthode `isEmpty` ?

Si oui, écrivez le corps de cette méthode. Sinon, justifiez votre réponse.

Partie 5 [1 point] Nommez le patron de conception utilisé par la classe `UnionView`.

2 Conversion de caractères [30 points]

Comme nous l'avons vu au cours, les 256 premiers caractères de la norme Unicode sont ceux de la norme ISO 8859-1. En conséquence, ces 256 caractères ont le même code, compris entre 0 et 255 (FF_{16}), dans ces deux normes. Par exemple, le code Unicode et ISO 8859-1 du caractère `H` — la lettre H majuscule — est 48_{16} ; celui du caractère `é` — la lettre E aigu minuscule — est $E9_{16}$; et ainsi de suite.

Le fait que ces caractères aient le même code dans les deux normes ne signifie pas que les séquences d'octets qui les représentent soient identiques. En effet, la norme ISO 8859-1 représente chaque caractère par un unique octet égal à son code, tandis que plusieurs encodages existent pour la norme Unicode. Le plus populaire d'entre eux, UTF-8, représente chaque caractère par un à quatre octets, en fonction de la plage à laquelle son code appartient. La table ci-dessous montre comment les caractères dont le code est compris entre 0 et $7FF_{16}$ sont encodés en UTF-8 — les caractères hors de cette plage n'étant pas utiles pour cet exercice :

Code	Code (base 2)	Octet 1	Octet 2
0 à $7F_{16}$	`gfedcba`	`0gfedcba`	
80_{16} à $7FF_{16}$	`kjihgfedcba`	`110kjihg`	`10fedcba`

Dans cette table, les bits pouvant être non nuls des codes des caractères sont représentés par des lettres minuscules : `a` pour le bit de poids le plus faible, `b` pour son voisin de gauche, etc. Ainsi, la première ligne de cette table signifie qu'un caractère Unicode dont le code est compris entre 0 et $7F_{16}$, et est donc représentable au moyen de 7 bits, est encodé en UTF-8 au moyen d'un seul octet dont le bit de poids le plus fort vaut 0, les 7 bits restants étant ceux du code du caractère.

Au moyen de ce qui précède, on peut déterminer que la séquence d'octets représentant la chaîne de caractères `Hé` est (en base 16) [48,E9] en ISO 8859-1, et [48,C3,A9] en UTF-8.

Le but de cet exercice est d'écrire des méthodes permettant de transformer une séquence d'octets représentant une chaîne de caractères encodée en ISO 8859-1 en la séquence d'octets représentant la même chaîne de caractères, mais encodée en UTF-8, et inversement. L'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre le fonctionnement de ces deux méthodes.

```
byte[] heyIso8859_1 = new byte[]{(byte) 0x48, (byte) 0xE9};
byte[] heyUtf8 = new byte[]{(byte) 0x48, (byte) 0xC3, (byte) 0xA9};

byte[] actualHeyUtf8 = iso8859_1ToUtf8(heyIso8859_1.clone());
assertArrayEquals(heyUtf8, actualHeyUtf8);

byte[] actualHeyIso8859_1 = utf8ToIso8859_1(heyUtf8.clone());
assertArrayEquals(heyIso8859_1, actualHeyIso8859_1);
```

Partie 1 [15 points] Écrivez la méthode `iso8859_1ToUtf8`, qui prend en argument un tableau d'octets correspondant à une chaîne de caractères encodée en ISO 8859-1 et retourne un tableau d'octets correspondant à la même chaîne mais encodée en UTF-8.

Pour vous simplifier la tâche, vous pouvez utiliser les classes `ByteArrayInputStream` et `ByteArrayOutputStream`, qui sont des flots permettant de lire et d'écrire des octets dans des tableaux d'octets. Nous ne les avons pas examinés en détail au cours, mais ils sont décrits dans le formulaire en annexe.

Vous n'avez pas le droit d'utiliser des méthodes de la bibliothèque Java qui transforment des caractères ou des chaînes en octets, ou inversement.

Partie 2 [15 points] Écrivez la méthode `utf8ToIso8859_1`, qui prend en argument un tableau d'octets correspondant à une chaîne de caractères encodée en UTF-8 et retourne un tableau d'octets correspondant à la même chaîne mais encodée en ISO 8859-1.

Si le tableau d'octets passé en argument ne correspond pas à une chaîne encodée en UTF-8, ou si au moins un des caractères ne peut pas être représenté en ISO 8859-1, votre méthode doit lever une `IllegalArgumentException`.

Vous n'avez pas le droit d'utiliser des méthodes de la bibliothèque Java qui transforment des caractères ou des chaînes en octets, ou inversement.

3 Patron *Observer* [35 points]

Les interfaces `Subject` et `Observer` ci-dessous sont des versions génériques de celles vues au cours, dans lesquelles le paramètre de type `T` représente le type de la valeur observée.

```
public interface Subject<T> {
  void addObserver(Observer<T> observer);
  void removeObserver(Observer<T> observer);
}
public interface Observer<T> {
  void update(T newValue);
}
```

Le but de cet exercice est d'écrire des classes représentant différents types de sujets, afin d'offrir certaines des fonctionnalités offertes par les valeurs observables de JavaFX.

Partie 1 [8 points] Écrivez une classe héritable nommée `AbstractSubject`, générique et implémentant l'interface `Subject`, fournissant des versions concrètes des méthodes `addObserver` et `removeObserver`, ainsi qu'une méthode protégée nommée `notifyObservers` prenant en argument la valeur à transmettre aux observateurs.

Afin que les sous-classes puissent avoir accès à la liste des observateurs, votre classe doit les stocker dans un attribut protégé.

Partie 2 [8 points] Écrivez une classe instanciable et générique nommée `Cell` représentant une cellule observable stockant une valeur d'un type quelconque et offrant :

— un unique constructeur prenant en argument la valeur initiale stockée dans la cellule,
— une méthode d'accès nommée `get` permettant d'obtenir la valeur actuellement stockée dans la cellule,
— une méthode de modification nommée `set` prenant en argument la nouvelle valeur à stocker dans la cellule et ne retournant rien.

`Cell` hérite de `AbstractSubject` afin d'être observable, et sa méthode `set` n'avertit ses observateurs que si la nouvelle valeur à stocker dans la cellule n'est pas égale (au sens de `equals`) à celle actuellement stockée. La valeur transmise aux observateurs est bien entendu la nouvelle valeur stockée dans la cellule.

L'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre le fonctionnement de cette classe.

```
Cell<String> cell = new Cell<>("a");
assertEquals("a", cell.get());

List<String> observedStrings = new ArrayList<>();
cell.addObserver(newString -> observedStrings.add(newString));

List.of("b", "c", "d").forEach(string -> cell.set(string));
assertEquals(List.of("b", "c", "d"), observedStrings);
assertEquals("d", cell.get());
```

Partie 3 [14 points] Écrivez une classe instanciable et générique nommée `MappedSubject` utilisant le patron *Decorator* pour obtenir un sujet qui soit le résultat de l'application d'une fonction à un sujet existant, à l'image de la méthode `map` des valeurs observables JavaFX, ou des flots Java.

L'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre le fonctionnement de cette classe.

```
Cell<Integer> integer = new Cell<>(0);
Subject<Integer> squared =
  new MappedSubject<>(integer, i -> i * i);
Subject<String> asStars =
  new MappedSubject<>(squared, count -> "*".repeat(count));

List<String> observedStars = new ArrayList<>();
asStars.addObserver(stars -> observedStars.add(stars));

List.of(1, 2, 3).forEach(i -> integer.set(i));
assertEquals(List.of("*", "****", "*********"), observedStars);
```

Votre mise en œuvre doit prendre garde à n'observer le sujet qu'on lui a donné que lorsqu'elle est elle-même observée, et elle ne doit jamais apparaître plus d'une fois dans sa liste d'observateurs. Elle peut par contre informer ses observateurs chaque fois que le sujet qu'on lui a donné informe les siens, sans se préoccuper de savoir si la nouvelle valeur qu'elle leur fournit diffère de l'ancienne.

Afin qu'une lambda puisse être utilisée pour spécifier la fonction à appliquer au sujet, vous pouvez utiliser l'interface fonctionnelle `Function` pour typer le second argument du constructeur de `MappedSubject`. Pour mémoire, cette interface appartient à la bibliothèque Java et est définie ainsi :

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

Partie 4 [5 points] Ajoutez à l'interface `Subject` une méthode par défaut nommée `map` permettant d'utiliser plus agréablement `MappedSubject`. Avec cette méthode, le début du test JUnit ci-dessus devrait pouvoir être récrit ainsi :

```
Cell<Integer> integer = new Cell<>(0);
Subject<String> asStars = integer
  .map(i -> i * i)
  .map(count -> "*".repeat(count));
// ... comme avant
```

4 Tableau dynamique [35 points]

La classe `SArrayList` vue au cours met en œuvre un tableau dynamique en stockant ses éléments dans un unique tableau Java qui est « redimensionné » (par copie) au besoin.

Une autre manière de mettre en œuvre un tel tableau dynamique consiste à stocker ses éléments dans plusieurs tableaux Java, nommés **morceaux** (*chunks*), dont la taille augmente de manière exponentielle. C'est-à-dire que le premier morceau, qui contient l'élément d'index 0, a une taille de 1 élément ; le second morceau, qui contient les éléments d'index 1 et 2, a une taille de 2 éléments ; le troisième morceau, qui contient les éléments d'index 3, 4, 5 et 6, a une taille de 4 éléments ; et ainsi de suite.

Le but de cet exercice est de compléter la classe `ChunkedArray` ci-dessous, qui représente un tableau dynamique de cette manière. Son attribut `size` contient le nombre d'éléments du tableau, tandis que son attribut `chunks` contient les morceaux.

```
public final class ChunkedArray<E> implements Iterable<E> {
  private int size = 0;
  private final E[][] chunks = (E[][]) new Object[Integer.SIZE - 1][];

  private static int chunkIndex(int index) { /* à faire */ }
  private static int indexInChunk(int index) { /* à faire */ }
  private static int chunkSize(int chunkIndex) { /* à faire */ }

  public int size() { return size; }
  public void addLast(E value) { /* à faire */ }
  public E get(int index) { /* à faire */ }
  public E set(int index, E value) { /* à faire */ }
  @Override
  public Iterator<E> iterator() { /* à faire */ }
}
```

Les morceaux sont créés à la demande, au fur et à mesure que la taille du tableau dynamique augmente. Initialement, le tableau dynamique est vide, et aucun morceau n'existe. Lorsque le premier élément y est ajouté au moyen de la méthode `addLast`, le premier morceau — de taille 1 — est créé, et l'élément ajouté y est stocké à l'index 0. Lorsque le second élément est ajouté au tableau dynamique, le second morceau — de taille 2 — est créé, et l'élément ajouté y est stocké à l'index 0. Lorsque le troisième élément est ajouté au tableau dynamique, il est stocké à l'index 1 du second morceau. Et ainsi de suite, le nombre maximum de morceaux étant de 31, ce qui implique que la taille maximale d'un tel tableau est de $2^{31} - 1$ éléments.

Dans un tel tableau, le morceau d'index i a une taille de 2^i éléments. Il est en conséquence facile de déterminer dans quel morceau, et à quel index dans ce morceau, se trouve un élément. Ainsi, l'élément d'index i du tableau dynamique se trouve :

— dans le morceau dont l'index est celui du bit à 1 le plus à gauche de la représentation en base 2 de $i+1$,
— à l'index, dans ce morceau, donné par les bits restants dans la représentation en base 2 de $i+1$ lorsqu'on a supprimé celui le plus à gauche.

Par exemple, sachant que $24+1 = 25_{10}$ s'écrit 11001_2 en base 2, on en conclut que l'élément d'index 24 du tableau dynamique se trouve :

— dans le morceau d'index 4, car le bit le plus à gauche de la représentation en base 2 de 25 est celui d'index 4,
— à l'index $1001_2 = 9_{10}$ dans ce morceau, qui a une taille de $2^4 = 16$ éléments.

Partie 1 [7 points] Écrivez les méthodes statiques suivantes :

— `chunkIndex` qui retourne l'index du morceau contenant l'élément d'index donné,
— `indexInChunk`, qui retourne l'index, dans le morceau auquel il appartient, de l'élément d'index donné,
— `chunkSize`, qui retourne la taille du morceau d'index donné.

Avant d'écrire votre réponse, consultez le formulaire en annexe pour voir les méthodes offertes par la classe `Integer`, qui simplifieront grandement vos mises en œuvre.

Partie 2 [8 points] Écrivez le corps de la méthode `addLast`, qui ajoute l'élément donné à la fin du tableau dynamique, en lui ajoutant si nécessaire un nouveau morceau.

Partie 3 [5 points] Écrivez le corps de la méthode `get`, qui retourne l'élément à l'index donné, ou lève une `IndexOutOfBoundsException` si celui-ci est invalide.

Partie 4 [7 points] Écrivez le corps de la méthode `set`, qui stocke l'élément donné à l'index donné du tableau dynamique, et retourne l'élément qui s'y trouvait précédemment ; lève une `IndexOutOfBoundsException` si l'index est invalide.

Partie 5 [8 points] Écrivez le corps de la méthode `iterator`, qui retourne un itérateur permettant de parcourir les éléments du tableau dynamique dans l'ordre, du premier au dernier. Cet itérateur ne doit pas donner la possibilité de supprimer un élément du tableau, donc sa méthode `remove` doit simplement lever une `UnsupportedOperationException`.

5 Projet [36 points]

Veuillez répondre aux questions suivantes dans le cahier de réponse qui vous a été fourni, et pas sur cette feuille. Attention, chaque réponse fausse annule une réponse correcte.

A : Points généralisés

L'algorithme MCTS utilise une notion de « points généralisés » pour évaluer les parties simulées. Listez les numéros des formules plus bas qui calculent des points généralisés équivalents à ceux utilisés dans le projet. C'est-à-dire les formules dont l'utilisation ne changerait pas le comportement de l'algorithme MCTS.

Dans ces formules, P_j représente les points généralisés du joueur j, r_j représente son rang, p_j représente son nombre de points et l_j représente le nombre de lignes horizontales complètes dans son mur. Toutes ces valeurs sont comprises entre 0 (inclus) et une valeur maximale, qui vaut par exemple 5 pour l_j.

Le complément d'une valeur x, noté $\overline{x}$, est égal à la valeur maximale moins la valeur en question. Par exemple, comme la valeur maximale de l_j vaut 5, si le joueur j possède deux lignes complètes dans son mur, alors $l_j = 2$ et $\overline{l_j} = 5 - 2 = 3$.

1. $P_j = \overline{r_j} \times 246 + p_j$
2. $P_j = 256 \times \overline{r_j} + p_j \times 3 + l_j$
3. $P_j = p_j + 256 \times \overline{r_j}$
4. $P_j = \overline{p_j} \times 256 + r_j$
5. $P_j = 256 \times r_j + p_j$
6. $P_j = 240 + 256 \times \overline{r_j} - \overline{p_j}$

B : Jeu aléatoire

Pour estimer la valeur des différents coups jouables, l'algorithme MCTS simule des parties durant lesquelles les joueurs jouent aléatoirement. Toutefois, il utilise des heuristiques pour éviter que les joueurs ne jouent de manière trop irréaliste, ce qui fausserait les estimations.

Admettons que lors d'une de ces simulations, l'algorithme MCTS doive faire jouer aléatoirement l'un des deux joueurs d'une partie dont l'état actuel est le suivant — sans les lignes de plancher, qui n'importent pas :

```
[1] A C DD  [2] A CC D  |  P1    . abCde  |  P2    A abCde
[3] A B EE  [4] BB C D  |       .. eabcd  |       E. eaBCd
[5] A BBB               |      C.. deabc  |      ... deaBc
                        |     CC.. cdeab  |     D... cdeab
[0] M                   |    B.... bcdea  |    ..... bcdea
```

Pour mémoire, les fabriques sont numérotées de 1 à 5, la zone centrale porte le numéro 0. Les cases du mur qui sont en majuscule contiennent une tuile de la couleur correspondante, celles qui sont en minuscule sont vides.

Les coups jouables par le joueur courant dans cet état peuvent être notés au moyen de la notation compacte dans laquelle la première lettre est l'index de la source, la seconde est la couleur des tuiles choisies, et la troisième l'index modifié de la destination, où 0 désigne la ligne plancher, 1 la première ligne de motif, 2 la seconde, et ainsi de suite.

Listez les numéros des coups ci-dessous qui pourraient être choisis par l'heuristique de l'algorithme MCTS si, dans l'état ci-dessus, il devait faire jouer aléatoirement le joueur `P1` :

10. `5B5`,
11. `1C3`,
12. `2C3`,
13. `1D2`.

Même question, mais en admettant que l'algorithme MCTS doive faire jouer le joueur `P2` :

15. `1D4`,
16. `1A0`,
17. `5A5`,
18. `3E2`.

C : Appariement des tuiles

Pour produire des animations cohérentes et agréables, Ajul utilise un algorithme qui apparie les emplacements de l'offre et de la demande dans un ordre donné, en six étapes successives.

Parmi les appariements ci-dessous, listez les numéros de ceux qui pourraient constituer la première (!) étape d'un algorithme d'appariement ayant les mêmes caractéristiques que celui utilisé dans le projet, c.-à-d. qui produise des animations cohérentes avec les règles du jeu. Faites l'hypothèse que l'offre et la demande sont triés de la même manière que dans le projet.

20. chaque élément de la demande hors du plateau est apparié avec le prochain élément de l'offre d'une source,
21. tous les éléments de la demande de la zone centrale qui le peuvent sont appariés avec ceux de l'offre d'une source,
22. chaque élément de la demande de plancher est apparié avec le prochain élément de l'offre d'une source,
23. chaque élément de la demande du mur est apparié avec l'élément correspondant de l'offre, c.-à-d. celui qui se trouve sur le plateau du même joueur, sur la ligne de motif correspondant à l'emplacement, dans la case la plus à droite.

D : Interface graphique

Parmi les affirmations suivantes à propos de l'interface graphique d'Ajul, listez les numéros de celles qui sont vraies :

30. Toute tuile visible à l'écran et qui n'est pas en déplacement se trouve exactement au-dessus d'une ancre.
31. Les ancres ne sont pas visibles à l'écran, car leur seul but est de faciliter le positionnement des tuiles.
32. Comme les sources de tuiles sont représentées par de simples rectangles grisés, sans cases individuelles, aucune ancre ne leur est associé.
33. Lorsqu'elles ne sont pas en déplacement, les tuiles visibles à l'écran ne se chevauchent jamais ; par contre, les tuiles qui ne sont pas visibles se chevauchent.
34. Pour placer une tuile au-dessus d'une ancre, il suffit de la placer aux mêmes coordonnées, les systèmes de coordonnées utilisés par les tuiles et les ancres étant identiques.
35. Lorsqu'une destination est survolée par des tuiles déplacées par un joueur humain, elle a accès à l'ensemble des coups valides que le joueur pourrait jouer avec ces tuiles, et peut l'utiliser pour déterminer si elle doit se mettre en évidence.

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 `Iterable` représente un objet itérable, c.-à-d. dont le contenu peut être parcouru au moyen d'un itérateur ou de la boucle *for-each*.

```
public interface Iterable<E> {
  // Retourne un itérateur sur les éléments du récepteur.
  Iterator<E> iterator();
}
```

Interface `Iterator`

L'interface `Iterator` représente un itérateur, c.-à-d. un objet permettant de parcourir les éléments d'une collection.

```
public interface Iterator<E> {
  // Retourne vrai ssi l'itérateur a encore un élément à fournir.
  boolean hasNext();

  // Retourne le prochain élément, ou lève NoSuchElementException s'il n'y en a plus.
  E next();
}
```

Interface `Collection`

L'interface `Collection` représente une collection. Les interfaces `List` (qui représente les listes) et `Set` (qui représente les ensembles) étendent cette interface.

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

  // Retourne la taille de la collection.
  int size();

  // Retourne vrai ssi la collection contient l'élément e.
  boolean contains(E e);

  // Ajoute l'élément e à la collection.
  void add(E e);

  // Supprime l'élément e de la collection, s'il s'y trouve.
  void remove(E e);
}
```

Classe `Integer`

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

```
public class Integer {
  // Le nombre de bits dans un entier de type int (32).
  public static final int SIZE;

  // Retourne le nombre de bits valant 0 se trouvant à gauche du bit valant 1
  // le plus à gauche de i, ou 32 si i vaut 0.
  // Par exemple, numberOfLeadingZeros(0b101) == 29.
  public static int numberOfLeadingZeros(int i);

  // Retourne un entier int ayant au plus un bit valant 1, à la même position que
  // le bit valant 1 le plus à gauche de i. Retourne 0 si i vaut 0.
  // Par exemple, highestOneBit(0b101) == 0b100.
  public static int highestOneBit(int i);
}
```

Classe `ByteArrayInputStream`

La classe `ByteArrayInputStream` représente un flot d'entrée d'octets dont le contenu provient d'un tableau.

```
public class ByteArrayInputStream extends InputStream {
  // Crée un flot d'entrée dont les octets proviennent du tableau array.
  public ByteArrayInputStream(byte[] array);

  // Lit et retourne le prochain octet sous la forme d'une valeur comprise entre 0
  // et 255 (inclus), ou –1 si la fin du flot a été atteinte.
  public int read();
}
```

Classe `ByteArrayOutputStream`

La classe `ByteArrayOutputStream` représente un flot de sortie d'octets qui écrit les octets qu'on lui fournit dans un tableau d'octets.

```
public class ByteArrayOutputStream extends OutputStream {
  // Crée un flot de sortie dont les octets sont écrits dans un tableau d'octets.
  public ByteArrayOutputStream();

  // Ecrit les 8 bits de poids faible de b dans le flot.
  public void write(int b);

  // Retourne un nouveau tableau contenant tous les octets ajoutés jusqu'à présent au flot.
  public byte[] toByteArray();
}
```