# Pratique de la programmation orientée-objet
# Examen final

3 juin 2016

Indications :

- l'examen dure de 12h15 à 16h00,
- 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 Nombres duaux [25 points]

En mathématiques, un **nombre dual** $z$ est un nombre ayant la forme suivante :

$$z = a + b\,\varepsilon$$

où $a$ et $b$ sont des nombres réels et $\varepsilon$ est un élément spécial ayant la propriété que $\varepsilon^2 = 0$. Les nombres duaux sont dès lors similaires aux nombres complexes, en ce qu'ils ont deux composantes et la seconde multiplie un élément doté d'une propriété particulière. Dans le cas des nombres complexes, cet élément est $i$ et sa propriété est que $i^2 = -1$, alors que dans le cas des nombres duaux, cet élément est $\varepsilon$ et sa propriété est que $\varepsilon^2 = 0$.

Comme pour les nombres complexes, il est possible de déterminer la définition des opérations arithmétiques de base sur les nombres duaux, en utilisant la propriété de l'élément spécial. Par exemple, l'addition, la soustraction et la multiplication de deux nombres duaux $z_1$ et $z_2$ peuvent se déterminer ainsi :

$$\begin{aligned}
z_1 + z_2 &= (a_1 + b_1\varepsilon) + (a_2 + b_2\varepsilon) = (a_1 + a_2) + (b_1 + b_2)\varepsilon \\
z_1 - z_2 &= (a_1 + b_1\varepsilon) - (a_2 + b_2\varepsilon) = (a_1 - a_2) + (b_1 - b_2)\varepsilon \\
z_1 \times z_2 &= (a_1 + b_1\varepsilon) \times (a_2 + b_2\varepsilon) = a_1a_2 + a_1b_2\varepsilon + b_1a_2\varepsilon + b_1b_2\varepsilon^2 \\
&= a_1a_2 + (a_1b_2 + b_1a_2)\varepsilon
\end{aligned}$$

D'autres opérations peuvent être déterminées de manière similaire.

Les nombres duaux rencontrent une utilisation intéressante dans une technique nommée dérivation algorithmique. Cette technique permet de calculer la dérivée d'une fonction quelconque pour une valeur donnée de son argument $x$, simplement en appliquant cette fonction au nombre dual $x + \varepsilon$. On peut illustrer cela au moyen de la fonction $f$ définie comme suit :

$$f(x) = 4x^3 + 5x^2 - x$$

La valeur de cette fonction appliquée à 2 se calcule facilement :

$$f(2) = 4 \cdot 2^3 + 5 \cdot 2^2 - 2 = 4 \cdot 8 + 5 \cdot 4 - 2 = 32 + 20 - 2 = 50$$

La valeur de la *dérivée* de cette fonction, $f'$, appliquée à 2, se calcule presque aussi aisément, en appliquant $f$ au nombre dual $2 + \varepsilon$ :

$$\begin{aligned}
f(2+\varepsilon) &= 4(2+\varepsilon)^3 + 5(2+\varepsilon)^2 - (2+\varepsilon) \\
&= 4(2+\varepsilon)(2+\varepsilon)(2+\varepsilon) + 5(2+\varepsilon)(2+\varepsilon) - (2+\varepsilon) \\
&= 4(4+4\varepsilon)(2+\varepsilon) + 5(4+4\varepsilon) - (2+\varepsilon) \\
&= 4(8+12\varepsilon) + (20+20\varepsilon) - (2+\varepsilon) \\
&= 32 + 48\varepsilon + 20 + 20\varepsilon - 2 - \varepsilon \\
&= 50 + 67\varepsilon
\end{aligned}$$

Ce résultat signifie que $f(2) = 50$ et $f'(2) = 67$. La validité de la seconde partie de ce résultat peut se vérifier en dérivant $f$ de manière symbolique et appliquant le résultat à 2. On obtient :

$$\begin{aligned}
f'(x) &= 12x^2 + 10x - 1 \\
f'(2) &= 12 \cdot 2^2 + 10 \cdot 2 - 1 = 48 + 20 - 1 = 67
\end{aligned}$$

La classe instanciable et immuable `Dual` ci-dessous représente un nombre dual, et le but de cet exercice est de compléter sa définition.

```
public final class Dual {
  private final double a, b;

  public Dual(double a, double b) { à faire }
  public Dual(double a) { à faire }

  public double a() { à faire }
  public double b() { à faire }

  public Dual add(Dual z2) { à faire }
  public Dual sub(Dual z2) { à faire }
  public Dual mul(Dual z2) { à faire }
}
```

**Partie 1 [4 points]** Ecrivez le corps des deux constructeurs. Le premier est le constructeur principal, le second est un constructeur secondaire qui construit un nombre dual avec 0 comme seconde composante.

Réponse :

**Partie 2 [2 points]** Ecrivez le corps des deux méthodes d'accès, `a` et `b`.

Réponse :

**Partie 3 [9 points]** Ecrivez le corps des méthodes `add`, `sub` et `mul` qui mettent en œuvre respectivement l'addition, la soustraction et la multiplication de deux nombres duaux, selon les définitions données plus haut.

Réponse :

**Partie 4 [6 points]** D'après-vous, les instances de la classe `Dual` doivent-elles être comparées par référence ou par structure ? Si vous pensez qu'elles doivent l'être par référence, justifiez votre réponse ; sinon, écrivez les redéfinitions des méthodes `equals` et `hashCode` correspondantes.

Réponse :

**Partie 5 [4 points]** L'extrait de programme ci-dessous utilise les nombres duaux pour calculer et imprimer la valeur de $f(2)$ et $f'(2)$ pour $f(x) = 4x^3 + 5x^2 - x$.

```
public final class Main {
  méthode f (à faire)

  public static void main(String[] args) {
    Dual fOf2 = f(new Dual(2, 1));
    System.out.println("␣f(2)␣=␣" + fOf2.a());
    System.out.println("f'(2)␣=␣" + fOf2.b());
  }
}
```

Complétez ce programme en donnant la définition de `f` sous la forme d'une méthode statique privée de la classe `Main`.

Réponse :

## 2 Moyenne glissante [28 points]

La **moyenne glissante** $a$ de taille $n$ d'une séquence de $k$ valeurs $v_1, v_2, \ldots, v_k$ est la moyenne des $n$ dernières valeurs :

$$a = \frac{1}{n} \sum_{i=0}^{n-1} v_{k-i}$$

Ces $n$ dernières valeurs sont généralement appelées la **fenêtre**, tandis que $n$ elle-même est appelée la **taille de la fenêtre**.

L'interface `MovingAverageIterator` ci-dessous représente un itérateur sur des valeurs de type `double` augmenté de la méthode `movingAverage`.

```
public interface MovingAverageIterator
    extends Iterator<Double> {
  public Optional<Double> movingAverage();
}
```

Pour une taille de fenêtre de $n$, la méthode `movingAverage` retourne soit :

- la valeur optionnelle vide, si moins de $n$ valeurs ont été produites par l'itérateur au moment de l'appel,
- une valeur optionnelle non vide contenant la moyenne des $n$ dernières valeurs produites par l'itérateur sinon.

Pour faciliter les choses, la méthode `remove` d'un tel itérateur peut simplement lever l'exception `UnsupportedOperationException`.

Ecrivez une classe nommée `MovingAverageIteratorAdapter` implémentant l'interface ci-dessus, et dont le constructeur prenne deux arguments :

1. un itérateur sur des valeurs de type `double`,
2. une taille de fenêtre.

Ce constructeur doit lever une `IllegalArgumentException` si la taille de la fenêtre n'est pas strictement positive. Sinon, l'instance construite doit produire exactement les même valeurs que l'itérateur passé à son constructeur, et doit correctement mettre en œuvre la méthode `movingAverage`. Par exemple, l'extrait de programme suivant :

```
List<Double> l = Arrays.asList(1.0, 2.0, 6.0, 4.0);
MovingAverageIterator i =
  new MovingAverageIteratorAdapter(l.iterator(), 3);
while (i.hasNext())
  System.out.println(i.next() + "␣/␣" + i.movingAverage());
```

doit afficher ceci :

```
1.0 / Optional.empty
2.0 / Optional.empty
6.0 / Optional[3.0]
4.0 / Optional[4.0]
```

Réponse :

## 3 Graphe de fonction [26 points]

La bibliothèque Java définit l'interface fonctionnelle `DoubleUnaryOperator`, une spécialisation de `UnaryOperator` à la situation où l'argument et le résultat de l'opérateur ont le type `double`. On peut le voir comme une version plus efficace de `UnaryOperator<Double>`. Cette interface est définie ainsi :

```
public interface DoubleUnaryOperator {
  public double applyAsDouble(double operand);
}
```

et est bien adaptée à la représentation de fonctions mathématiques des réels vers les réels ($\mathbb{R} \to \mathbb{R}$).

La classe `FunctionComponent` ci-dessous est un composant Swing capable d'afficher le graphe d'une telle fonction. Son constructeur prend en arguments :

- la fonction `f` dont le graphe doit être dessiné,
- le cadre (`minX`, `maxX`, `minY` et `maxY`) du graphe à dessiner, exprimé dans le repère du plan.

```
public final class FunctionComponent extends JComponent {
  private final DoubleUnaryOperator f;
  private final double minX, maxX, minY, maxY;

  public FunctionComponent(DoubleUnaryOperator f,
                           double minX, double maxX,
                           double minY, double maxY) {
    this.f = f;
    this.minX = minX; this.maxX = maxX;
    this.minY = minY; this.maxY = maxY;
  }
  @Override
  protected void paintComponent(Graphics g) { à faire }
}
```

Par exemple, une instance de ce composant pourrait être créée ainsi afin de dessiner le graphe de la fonction $f(x) = x^2$ pour $x \in [-5;5]$ et $y \in [-1;20]$ :

```
new FunctionComponent(x -> x * x, -5, 5, -1, 20);
```

En plaçant ce composant dans une fenêtre affichée à l'écran, on devrait obtenir un résultat similaire à celui de la figure 1.

Ecrivez le corps de la méthode `paintComponent` ci-dessus. Vous pouvez faire l'hypothèse que la couleur et la largeur du trait sont déjà corrects. Le graphe doit être dessiné sous la forme d'une séquence de exactement $w-1$ sements de droite, où $w$ est la largeur du composant retournée par sa méthode `getWidth`. Dans le repère du composant, le premier segment de droite doit avoir 0 comme coordonnée $x$ de départ, et 1 comme coordonnée $x$ d'arrivée ; le second segment doit avoir 1 comme coordonnée $x$ de départ, 2 comme coordonnée $x$ d'arrivée ; et ainsi de suite. Ces segments de droite doivent être dessinés au moyen de la méthode `drawLine` du contexte graphique (classe `Graphics`), dont la signature est :

```
public void drawLine(int x1, int y1, int x2, int y2) {...}
```

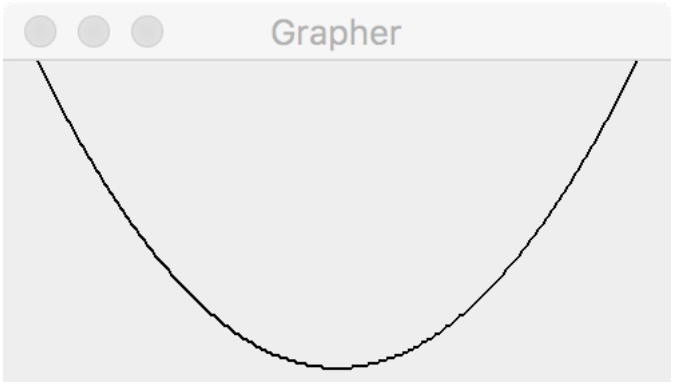


Fig. 1: Graphe de $f(x) = x^2$ entre $-5$ et 5

où (`x1`, `y1`) sont les coordonnées de l'une des extrémités du segment, et (`x2`, `y2`) les coordonnées de l'autre extrémité.

Réponse :

# 4 Sérialisation d'entiers [22 points]

En Java, les entiers de type `int` font 32 bits et peuvent dès lors toujours être sérialisés comme une séquence de 4 octets. Néanmoins, lorsqu'un grand nombre d'entiers doit être sérialisé et la plupart d'entre eux sont positifs et proches de zéro, cette technique est coûteuse en mémoire. Il est possible de faire mieux en utilisant une technique de sérialisation à longueur variable.

L'idée est la suivante : premièrement, 3 bits nuls, dits *de remplissage*, sont ajoutés en tête des 32 bits de l'entier à sérialiser — conceptuellement en tout cas —, afin d'obtenir un total de 35 bits. Ces 35 bits sont découpés en cinq groupes de 7 bits et tous les groupes de tête constitués exclusivement de zéros sont ignorés, à l'exception du dernier dans le cas où *tous* les groupes ne contiennent que des zéros. Finalement, les groupes restants sont sérialisés du *moins* significatif au *plus* significatif.

Chaque groupe est sérialisé au moyen d'un octet dont les 7 bits de poids faible sont ceux du groupe, tandis que son bit de poids fort vaut 1 pour tous les groupes sauf le dernier sérialisé. Dès lors, le bit de poids fort indique si d'autres octets suivent pour ce même entier, ou non.

Par exemple, la valeur `int` représentant l'entier 71 est formée des 32 bits :

```
00000000000000000000000001000111
```

Après ajout des 3 bits de remplissage et découpage en groupes de 7 bits, on obtient :

```
000 0000000 0000000 0000000 0000000 1000111
```

où les bits soulignés sont ceux de remplissage. Les quatre groupes de tête étant constitués exclusivement de zéros, ils sont ignorés. L'unique groupe restant est sérialisé sous la forme d'un seul octet dont les 7 bits de poids faible sont ceux du groupe (`1000111`) et dont le bit de poids fort vaut 0 étant donné qu'il s'agit du dernier groupe. Le résultat est donc l'unique octet suivant :

```
01000111
```

Un autre exemple est celui de l'entier 14 465, qui correspond aux cinq groupes suivants :

```
000 0000000 0000000 0000000 1110001 0000001
```

Une fois les trois groupes de tête ignorés, les deux derniers sont sérialisés sous la forme d'une séquence de deux octets. Le premier contient les 7 bits les *moins* significatifs (`0000001`) et son bit de poids fort vaut 1 pour indiquer qu'un autre groupe suit. Le second octet contient l'autre groupe (`1110001`) et son bit de poids fort vaut 0 étant donné qu'il s'agit du dernier groupe. Le résultat est donc la séquence de deux octets suivante :

```
10000001 01110001
```

La classe non instanciable `VarIntSerializer` ci-dessous représente un (dé)sérialiseur d'entiers `int` utilisant la technique décrite ci-dessous. Le but de cet exercice est de compléter sa définition.

```
public final class VarIntSerializer {
  private VarIntSerializer() {}
  public static void serialize(ByteBuffer b, int i) { à faire }
  public static int deserialize(ByteBuffer b) { à faire }
}
```

**Partie 1 [11 points]** Ecrivez le corps de la méthode `serialize`, qui sérialise l'entier reçu au moyen de la technique décrite plus haut, en écrivant le résultat dans le tampon d'octets donné au moyen d'appels à la méthode `put` relative.

Réponse :

**Partie 2 [11 points]** Ecrivez le corps de la méthode `deserialize`, qui désérialise et retourne le prochain entier du tampon d'octets donné, en lisant le nombre d'octets nécessaire au moyen d'appels à la méthode `get` relative.

Réponse :

## 5 Construction de niveau [10 points]

Dans XBlast, les niveaux sont représentés par des instances de la classe `Level`. Le but de cet exercice est de compléter la définition de la méthode `myLevel` ci-dessous afin qu'elle construise un niveau dont l'état initial est celui présenté à la figure 2.

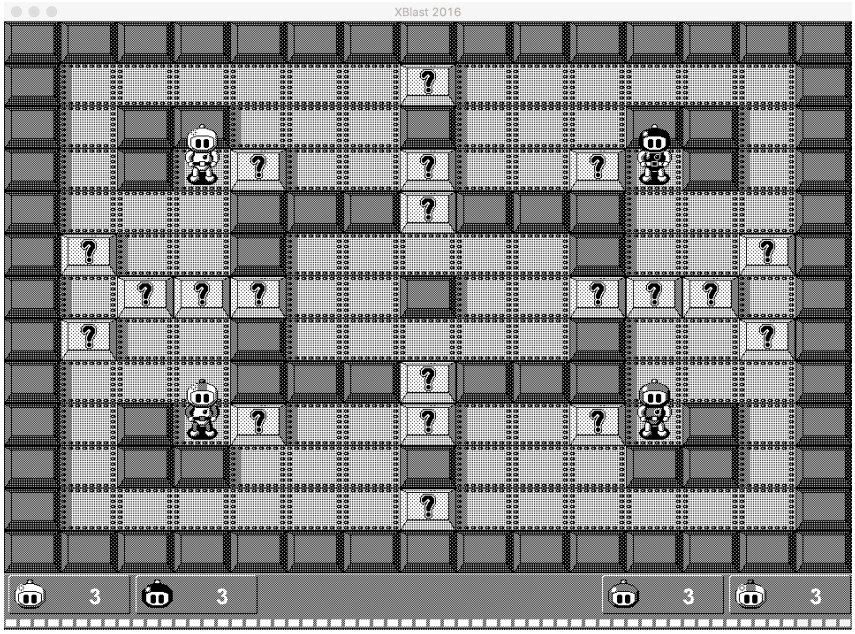


Fig. 2: Un nouveau niveau

```
public final class MyLevel {
  public static Level myLevel() {
    BoardPainter painter = omis (peintre standard) ;

    Block __ = Block.FREE;
    Block XX = Block.INDESTRUCTIBLE_WALL;
    Block oo = Block.DESTRUCTIBLE_WALL;

    List<List<Block>> blocks = à faire ;
    Board board = Board.ofQuadrantNWBlocksWalled(blocks);

    int l = 3, b = 2, r = 3;
    List<Player> players = Arrays.asList(
      new Player(PLAYER_1, l, new Cell( à faire ), b, r),
      new Player(PLAYER_2, l, new Cell( à faire ), b, r),
      new Player(PLAYER_3, l, new Cell( à faire ), b, r),
      new Player(PLAYER_4, l, new Cell( à faire ), b, r));

    return new Level(painter, new GameState(board, players));
  }
}
```

**Partie 1 [6 points]** Ecrivez la définition de la variable `blocks` qui permet d'obtenir le plateau de la figure 2 lorsqu'on la passe à `ofQuadrantNWBlocksWalled`.

Réponse :

**Partie 2 [4 points]** La variable `players` contient l'état initial des joueurs. Dans le code ci-dessus, leurs positions initiales ont été omises. Donnez les arguments à passer au constructeur de `Cell` afin que ces positions initiales soient celles de la figure 2 :

- pour le joueur 1 : `new Cell(_____, _____)`,
- pour le joueur 2 : `new Cell(_____, _____)`,
- pour le joueur 3 : `new Cell(_____, _____)`,
- pour le joueur 4 : `new Cell(_____, _____)`.

# 6 Explosions [14 points]

Lorsqu'une bombe explose dans XBlast, elle produit une explosion. Dans notre mise en œuvre, une telle explosion est représentée par une séquence de séquences de cases, créée par la méthode `explosion` de la classe `Bomb`. La classe ci-dessous produit des explosions différentes de celles des bombes « normales » du projet.

```
public final class Bomb {
  private final Cell pos; // position de la bombe

  // autres attributs et méthodes omis

  public List<Sq<Sq<Cell>>> explosion() {
    List<Sq<Sq<Cell>>> explosion = new ArrayList<>();
    for (Direction d: Direction.values()) {
      Sq<Cell> p =
        Sq.iterate(pos, q -> q.neighbor(d)).limit(3);
      Sq<Sq<Cell>> a1 =
        Sq.repeat(d.ordinal(), Sq.repeat(1, pos));
      Sq<Sq<Cell>> a2 = Sq.repeat(4, p);
      explosion.add(a1.concat(a2));
    }
    return Collections.unmodifiableList(explosion);
  }
}
```

Pour mémoire, l'énumération `Direction` est définie ainsi :

```
public enum Direction { N, E, S, W }
```

Supposons qu'une bombe du type ci-dessus soit placée sur la case centrale du plateau de $7 \times 7$ cases de la figure 3, et explose de manière à émettre ses premières particules au coup d'horloge $t$. A ce coup d'horloge, toutes les particules occupent la case centrale, qui est coloriée pour indiquer cela. Complétez la figure en coloriant toutes les cases occupées par au moins une particule d'explosion aux coups d'horloge $t+1$ à $t+7$.

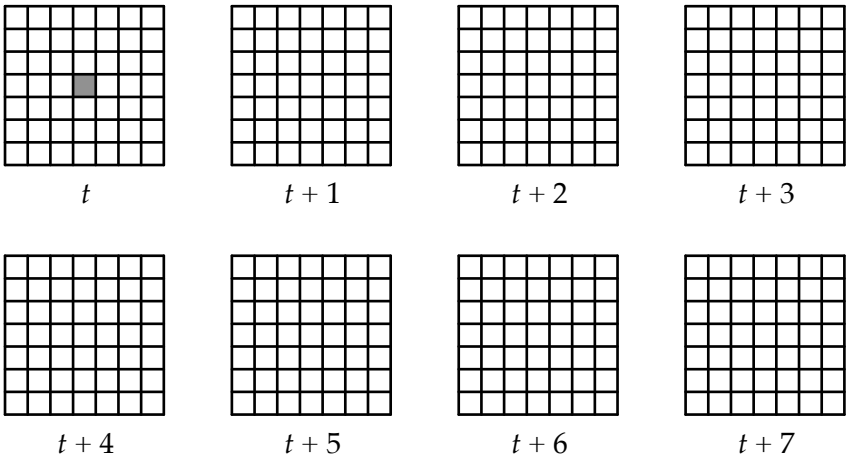


Fig. 3: Particules d'explosion

# Formulaire

Ce formulaire présente toutes les parties de la bibliothèque Java nécessaires à cet examen. De nombreuses méthodes inutiles ont été omises.

## Interface `Iterator`

L'interface `java.util.Iterator` représente un itérateur, c-à-d un objet permettant le parcours d'une collection d'autres objets.

```
interface Iterator<E> {
  // Retourne vrai ssi cet itérateur peut encore livrer des éléments.
  boolean hasNext();

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

  // Supprime la valeur retournée par le dernier appel à next, ou lève
  // l'exception IllegalStateException si next n'a pas encore été
  // appelée, ou si remove est appelée deux fois de suite.
  void remove();
}
```

## Interface `Iterable`

L'interface `java.lang.Iterable` représente les objets itérables, c-à-d ceux dont le contenu peut être parcouru au moyen d'un itérateur ou de la boucle *for-each*.

```
interface Iterable<E> {
  // Retourne un itérateur sur les éléments de l'objet.
  Iterator<E> iterator();
}
```

## Interface `Queue`

L'interface `java.util.Queue` représente les queues. Elle est implémentée, entre autres, par la classe `ArrayDeque`.

```
interface Queue<E> extends Iterable<E> {
  // Retourne le nombre d'éléments contenus dans la queue.
  int size();

  // Ajoute l'élément e à la fin de la queue et retourne vrai.
  boolean add(E e);

  // Supprime et retourne l'élément du début de la queue.
  E remove();

  // Retourne un itérateur sur les éléments de la queue.
  Iterator<E> iterator();
}
```

### Classe `Optional`

La classe `java.util.Optional` représente une valeur optionnelle.

```
class Optional<E> {
  // Crée et retourne une valeur optionnelle vide.
  static <E> Optional<E> empty();

  // Crée et retourne une valeur optionnelle contenant la valeur donnée.
  static <E> Optional<E> of(E e);
}
```

### Classe `Objects`

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

```
class Objects {
  // Retourne une valeur de hachage pour les valeurs données.
  static int hash(Object... values);
}
```

### Classe `ByteBuffer`

La classe `java.nio.ByteBuffer` représente un tampon d'octets de taille fixe doté d'une position qui permet la lecture/écriture relative d'octets.

```
class ByteBuffer {
  // Retourne l'octet à la position courante, puis incrémente cette dernière.
  byte get();

  // Ecrit l'octet b à la position courante, puis incrémente cette dernière.
  void put(byte b);
}
```

### Classe `JComponent`

La classe `javax.swing.JComponent` représente un composant Swing.

```
class JComponent {
  // Retourne la largeur du composant.
  int getWidth();

  // Retourne la hauteur du composant.
  int getHeight();
}
```