Héritage, classes abstraites et interfaces

Objectif

Cet exercice consiste à implémenter la mécanique derrière une fenêtre d'affichage représentant divers types de ballons qui se déplacent au fil du temps. Attention à ne pas vous tromper sur le but de cette leçon : le sujet principal n'est pas l'affichage de graphiques mais bien l'héritage entre les classes !

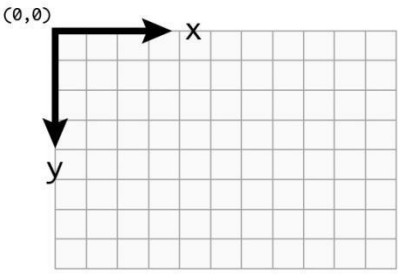
Partie graphique

Toute la partie graphique est assurée par trois classes dont le code vous est donné :

- `FenêtreGraphique`, qui est la classe principale représentant la fenêtre (titre et bordures comprises) où les ballons vont s'afficher ;
- `PlanEnvol`, qui s'occupe plus particulièrement de l'espace où les ballons sont affichés (l'intérieur de la fenêtre) ;
- `Cercle`, qui est la traduction en termes graphiques de l'affichage d'un ballon (pour simplifier les choses, chaque ballon sera représenté par un simple cercle).

Dans un premier temps, lisez le code de la classe `Cercle` (qui est une classe très simple). Vous verrez que chaque ballon à représenter est affiché sous la forme d'un cercle repéré par les attributs suivants :

- la position de son centre (`posX`, `posY`) mesurée en pixels par rapport au coin supérieur gauche de la zone d'affichage ;
  Attention : contrairement aux mathématiques, où on a l'habitude de mesurer la composante « y » en partant de 0 en bas, en informatique, le niveau « y = 0 » correspond généralement au dessus de la zone d'affichage (voir représentation ci-contre). Un ballon qui se déplace vers le haut verra donc sa coordonnée y diminuer !



- son rayon (`rayon`) mesuré en pixels ;
- la couleur de son bord (`couleurBord`) et de son intérieur (`couleurDisque`). Ces deux couleurs sont exprimées en utilisant le type standard `Color`, qui comporte toute une série de constantes (en fait, des attributs statiques/de classe) s'écrivant `Color.BLACK`, `Color.BLUE`, `Color.GREEN` etc. (tapez `Color.` puis utilisez l'autocomplétion de votre IDE pour voir les différentes possibilités).

Vous n'aurez pas à utiliser directement la classe `PlanEnvol` mais bien `FenêtreGraphique`. Voici ce qu'il faut savoir à propos de cette dernière :

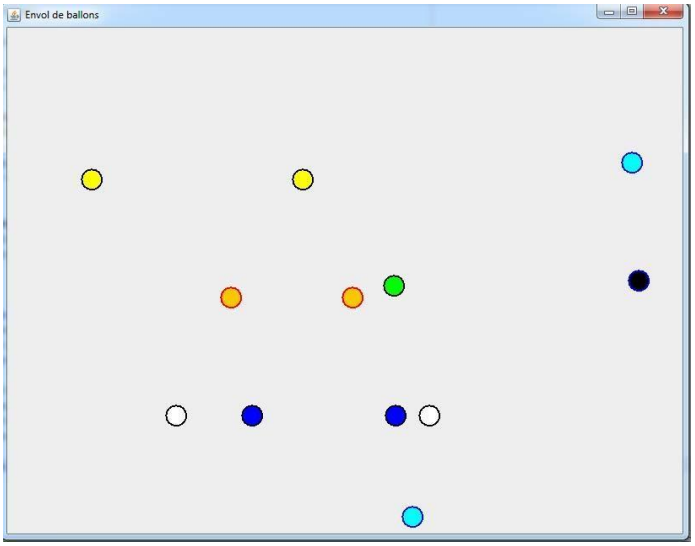
- Pour créer une fenêtre d'affichage, il faut utiliser le constructeur suivant :
  ```
  public FenêtreGraphique (int longueur, int hauteur)
  ```
  où `longueur` et `hauteur` sont les dimensions du cadre d'affichage en pixels.

  Dans le cadre de cet exercice, fixons une fois pour toute la taille de la zone d'affichage à 800 pixels de long et 600 pixels de haut (si cette taille ne vous convient pas, vous pouvez la modifier, mais cet énoncé se basera sur ces valeurs).
- Une fois une fenêtre graphique créée, vous pourrez lui ajouter un à un les ballons à afficher, en utilisant la méthode suivante.
  ```
  public void ajouteBallon (Ballon b)
  ```
- Finalement, une fois tous les ballons ajoutés, pour déclencher l'affichage et l'exécution du programme (qui fera évoluer les ballons à l'écran), il vous suffira d'appeler la méthode suivante sur la fenêtre graphique.
  ```
  public void go ()
  ```

Un premier ballon

Créez un nouveau projet et incorporez les trois classes prédéfinies (`FenêtreGraphique`, `PlanEnvol` et `Cercle`) dans un package appelé `ballons`. C'est normal s'il y a des erreurs : c'est parce que le code fait appel à une classe `Ballon` qui n'est pas encore définie… mais vous allez y remédier !

Définissez une nouvelle classe appelée `Ballon`. Chaque ballon commencera sa course sur le bord inférieur de la fenêtre et, au fil du temps, montera en ligne droite vers le bord supérieur. Cela signifie que le ballon partira d'une position (x,y) où $0 \leq x \leq 800$ et $y = 600$ (souvenez-vous que $y = 0$ correspond au bord supérieur). Puis, peu à peu, alors que la position x restera constante, la position y, elle, diminuera. Pour fixer les idées, disons que y diminuera de 1 à chaque itération / mise à jour (en anglais, on parle souvent de « *tick* »).



Dans la classe `Ballon`, créez

- un **constructeur** permettant d'indiquer la position x où le ballon est lâché ;
- une **méthode** `void avance()` qui mettra le ballon à jour à chaque *tick* (c'est-à-dire qui modifiera sa position) ;
- une **méthode** `Cercle getCercle()` qui sera automatiquement appelée par les classes graphiques et qui devrait « traduire » le ballon sous forme d'un objet `Cercle` (par défaut, on représentera les ballons par des cercles noirs avec un intérieur blanc ayant un rayon de 12 pixels).

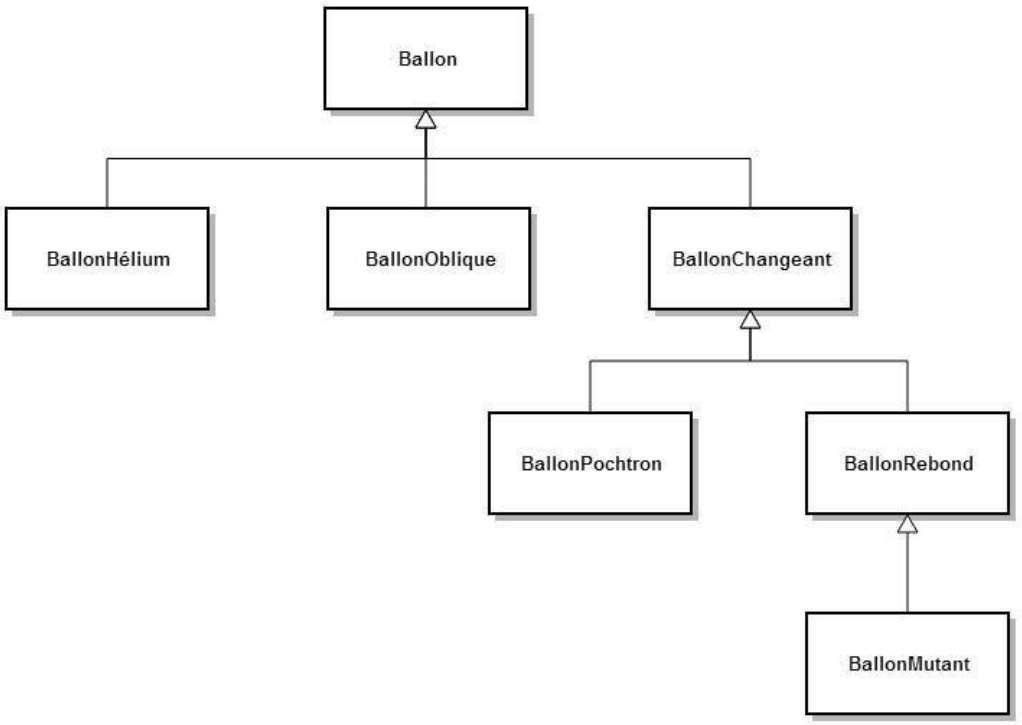
Pour tester votre classe `Ballon`, utilisez le code suivant dans la méthode principale :

```
FenêtreGraphique fen = new FenêtreGraphique (800, 600);
fen.ajouteBallon(new Ballon (200));
fen.go();
```

À l'exécution, cela devrait faire apparaître un ballon qui se déplace de bas en haut et situé au quart de la largeur de la fenêtre. Ajoutez d'autres ballons à votre gré !

Une histoire d'héritage

En plus des simples ballons décrits ci-dessus, vous allez ajouter diverses autres classes correspondant à des ballons dont les comportements sont différents. Comme il s'agira toujours de ballons (c'est-à-dire, dans ce cadre-ci, de cercles se déplaçant sur l'écran), une bonne partie du code pourra être réutilisée (même si, pour que cela soit possible, il faudra parfois modifier le code de `Ballon`). La hiérarchie des classes à construire est résumée dans le diagramme UML ci-dessous.



BallonHélium et BallonOblique

Dans un premier temps, créez une nouvelle classe appelée `BallonHélium` et descendant de `Ballon`. Un ballon gonflé à l'hélium est jaune avec un bord noir ; il grimpe dans les airs trois fois plus vite qu'un ballon normal !

Faites de même pour `BallonOblique`. Un ballon oblique est un ballon bleu avec un bord noir qui ne monte pas en ligne droite mais en oblique, soit vers la droite, soit vers la gauche (la direction est choisie au moment où le ballon oblique est créé, par exemple avec un argument booléen pour le constructeur).

1. Tout d'abord, créez des **constructeurs** pour chacune de ces deux classes. Souvenez-vous que, dans une sous-classe, la première opération que le constructeur doit faire, c'est un appel au constructeur de la classe mère (cet appel se fait via le mot clef `super` ; dans certains cas, il n'est pas nécessaire de mentionner l'appel explicitement — voir cours de théorie).

2. Vous allez sans doute vous rendre compte à un moment ou à un autre que vous avez besoin d'accéder à des attributs privés. Comme vous ne pouvez pas le faire directement, vous allez sans doute devoir modifier le code de la classe `Ballon` pour y ajouter des getters et des setters.

   Notez que les getters/setters ne doivent pas forcément être « standards ». Ainsi, vu qu'il s'agit de ballons qui se déplacent, on pourrait envisager de coder un setter modifiePositionX(dx) *qui ajoute dx à la position x plutôt qu'un setter standard* setPositionX(nouvX)*, voire même un double setter* modifiePosition(dx,dy) *qui permet de modifier les deux coordonnées en une seule fois.*

3. Les ballons gonflés à l'hélium fonctionnent grosso modo comme les ballons normaux. Par contre, dans le cas des ballons obliques, il faudra mémoriser leur direction. Cela ne pose pas de problème : les classes-filles héritent de tous les attributs de la classe-mère et peuvent en ajouter de nouveau. Réfléchissez à la meilleure manière de stocker cette information !

4. Tous les ballons sont représentés par des cercles de 12 pixels de rayon. D'après le principe du « point de modification unique », cela signifie que le nombre 12 ne devrait être indiqué qu'à un seul endroit (dans la classe `Ballon` sans doute). Arrangez-vous pour que ce soit bien le cas ! De même, si on décide de modifier la manière dont les ballons sont représentés (en utilisant une autre classe que `Cercle`), cette modification ne devrait toucher qu'un seul endroit du code. La classe `Ballon` devrait donc être la seule à faire un appel explicite à `new Cercle`.

   Concrètement, cela signifie que la méthode `getCercle` devrait être héritée de `Ballon` sans être redéfinie. Pour que cela puisse fonctionner, le code doit être suffisamment général. Ainsi, tant qu'il n'y avait que la classe `Ballon`, l'implémentation suivante suffisait.

   ```
   public Cercle getCercle () {
    return new Cercle (posX, posY, 12, Color.BLACK, Color.WHITE);
   }
   ```

   Mais, pour que le même code puisse être utilisé pour toutes les classes-filles, il faut faire en sorte que les arguments `posX`, `posY`, `Color.BLACK` et `Color.WHITE` puissent être modifiés. Deux moyens pour ce faire (on peut aussi combiner ces méthodes) :

   (a) utiliser des attributs privés et, dans les classes-filles, les modifier via des setters :

   ```
   new Cercle (posX, posY, 12, couleurBord, couleurDisque)
   ```

   (b) faire appel à des méthodes qui pourront être redéfinies dans les classes filles :

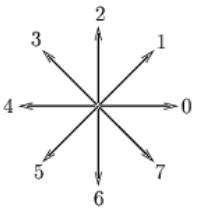
   ```
   new Cercle (getPosX(), getPosY(), 12, getCouleurBord(), getCouleurDisque())
   ```

BallonChangeant

Jusqu'ici, tous les ballons que nous avons définis avaient un mode de déplacement unique : vers le haut (plus ou moins vite) ou en oblique mais toujours dans la même direction. Les ballons changeants (pensez à des ballons percés par exemple) ne fonctionnent pas de la même manière : ils partent dans une certaine direction et, aléatoirement, de temps en temps, changent de direction.

Comme ces ballons peuvent aller dans n'importe quelle direction, ils commencent leur trajet à mi-hauteur de la fenêtre (c'est-à-dire à la position $y = 300$). Ce sont des ballons verts à bord noir.

La direction initiale de ces ballons est déterminée aléatoirement : le ballon peut se déplacer d'au plus 1 pixel sur chacun des axes (il y a donc 8 directions possibles : voir diagramme ci-contre). Pour calculer la direction, vous pouvez utiliser la méthode statique `Math.random()` qui fournit une valeur aléatoire v telle que $0 \leq v < 1$ (il y a plusieurs manières d'utiliser cette méthode… à vous de réfléchir à la solution la plus judicieuse).



À chaque *tick*, il y a 10% de chances pour que le ballon changeant change de direction. Si ce n'est pas le cas, il continue à progresser dans la même direction qu'au *tick* précédent.

Dans un premier temps, réalisez une implémentation simple pour la classe `BallonChangeant` et testez-la.

Dans un second temps, lisez la présentation des classes `BallonPochtron` et `BallonRebond` (qui hériteront de `BallonChangeant`) et revisitez le code de `BallonChangeant` pour faciliter leur future implémentation. Lorsque vous avez écrit les classes `BallonHélium` et `BallonOblique`, vous avez été amenés à modifier le code de `Ballon` pour le rendre plus générique ; ici, l'exercice consiste à rendre `BallonChangeant` suffisamment générique *dès le départ*, en prévoyant ce qui sera utile ou nécessaire à l'implémentation des classes-filles. (Bien sûr, si vous vous rendez compte par la suite que vous avez oublié quelque chose, rien ne vous empêchera de modifier le code à ce moment-là).

BallonPochtron et BallonRebond

Les ballons pochtrons et les ballons à rebonds sont des ballons changeants : ils ne se déplacent pas toujours dans la même direction.

Le ballon pochtron, intérieur orange et bord rouge, commence son parcours sur le bord inférieur de la fenêtre. Peu à peu, il se déplace vers le bord supérieur… mais pas vraiment en ligne droite : il zigzague. À chaque *tick*, il monte de 2 pixels (il avance vite). Horizontalement, soit il reste à la même position, soit il va 1 pixel vers la droite, soit il va 1 pixel vers la gauche (ce choix varie à chaque *tick*).

Le ballon à rebond, intérieur cyan et bord bleu, quant à lui, commence son parcours à mi-hauteur de la fenêtre. Il se déplace toujours en ligne droite et ne change de direction que quand il sort des limites de la fenêtre d'affichage. Sa direction est déterminée aléatoirement : à chaque *tick*, ses coordonnées x et y peuvent se déplacer d'au plus 5 pixels. Une fois sa direction déterminée, il la conserve jusqu'à ce qu'il se retrouve hors-champ. S'il sort à gauche ou à droite (c'est-à-dire $x < 0$ ou $x > 800$), sa direction s'inverse dans le sens des x. S'il sort en haut ou en bas (c'est-à-dire $y < 0$ ou $y > 600$), sa direction s'inverse dans le sens des y.

Concrètement, si le ballon à rebond se déplace à chaque *tick* en faisant +3 en x et -2 en y, au moment où il sortira à droite de l'écran ($x > 800$), il changera de direction : dorénavant, il fera -3 en x et -2 en y (sa direction s'est inversée dans le sens des x). Si, ensuite, il sort par le haut de la fenêtre ($y < 0$), sa direction s'inversera dans le sens des y : il fera -3 en x et +2 en y.

Conseil. Comme indiqué plus haut, on vous demande tout d'abord de revisiter le code de `BallonChangeant` *pour « préparer le terrain » et pouvoir plus facilement coder* `BallonPochtron` *et* `BallonRebond`*.*

BallonMutant [dépassement ?]

Finalement, parmi les ballons à rebonds, certains sont mutants dans le sens où, à chaque *tick* où le ballon ne change pas de direction (c'est-à-dire quand il n'est pas hors-cadre), il y a 0,5% de chances pour que le ballon se dédouble. Un second ballon est alors créé au même endroit (mais avec une direction de déplacement choisie aléatoirement). Les ballons mutants sont noirs à bord bleu (et peuvent rapidement inonder le terrain d'envol !).

Dépassement dans le dépassement. Après avoir donné naissance à un nouveau ballon mutant, le ballon mutant perd de sa vitalité : il devient gris foncé. Un ballon mutant gris foncé peut encore donner naissance à un autre ballon mutant mais, après avoir fait cela, il devient définitivement inerte (et gris clair). Un ballon mutant gris clair se contente de rebondir mais ne peut plus engendrer d'autres ballons mutants.

Code des trois classes prédéfinies

FenêtreGraphique

```
package ballons;

import java.awt.*;
import java.awt.event.*;
import javax.swing.JFrame;
import javax.swing.Timer;
public class FenêtreGraphique extends JFrame implements ActionListener { PlanEnvol plan;
    public FenêtreGraphique(int longueur, int hauteur) {
        super("Envol de ballons");
        plan = new PlanEnvol(longueur, hauteur);
        add(plan);
        pack();

        Dimension tailleÉcran = Toolkit.getDefaultToolkit().getScreenSize();
        Dimension tailleFenêtre = getSize();
        int espaceGauche = (tailleÉcran.width - tailleFenêtre.width) / 2; int
        espaceHaut  =  (tailleÉcran.height  -  tailleFenêtre.height)  /  2;
        setLocation(espaceGauche, espaceHaut);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public void ajouteBallon(Ballon b) {
        plan.ajouteBallon(b);
    }

    public void actionPerformed(ActionEvent e) {
        plan.maj();
    }

    public void go() {
        plan.repaint();
        plan.setRunning();
        setVisible(true);
        Timer maj = new Timer(10, this);
        maj.start();
    }
}
```

Cercle

```
package ballons;

import java.awt.Color;

public class Cercle {
    public int posX, posY, rayon;
    public Color couleurBord, couleurDisque;

    public Cercle(int posX, int posY, int rayon, Color couleurBord, Color couleurDisque) {
        this.posX = posX;
        this.posY = posY;
        this.rayon = rayon;
        this.couleurBord = couleurBord;
        this.couleurDisque = couleurDisque;
    }
}
```

PlanEnvol

```
package ballons;

import java.awt.*; import
java.awt.geom.*; import
java.util.ArrayList; import
javax.swing.JPanel;

public class PlanEnvol extends JPanel {
    private ArrayList<Ballon> ballons = new ArrayList<>();
    private boolean running = false;
    private ArrayList<Ballon> ballonsAAjouter = new ArrayList<>();

    public PlanEnvol(int longueur, int hauteur) {
         setPreferredSize(new Dimension(longueur, hauteur));
    }

    public void ajouteBallon(Ballon b) {
        if (!running) {
            ballons.add(b);
        } else {
            ballonsAAjouter.add(b);
        }
    }
     public void setRunning() {
         running = true;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2D = (Graphics2D) g;
        for (Ballon b : ballons) {
            Cercle cercle = b.getCercle();
            Ellipse2D cercle2D = new Ellipse2D.Float(cercle.posX - cercle.rayon,
                                                    cercle.posY - cercle.rayon, cercle.rayon * 2, cercle.rayon * 2);
            g2D.setPaint(cercle.couleurDisque);
            g2D.fill(cercle2D); g2D.setStroke(new
            BasicStroke(2));
            g2D.setPaint(cercle.couleurBord);
            g2D.draw(cercle2D);
        }

    }

    public void maj() {
        for (Ballon b : ballons) {
            b.avance();
        }
        for (Ballon b : ballonsAAjouter) {
            ballons.add(b);
        }
        ballonsAAjouter.clear();
        repaint();
    }
}
```