Programmation orientée objet avancée
- Série 4 -

Objectifs

- Manipuler l'objet Graphics
- Créer et lancer des threads travaillant en parallèle
- Créer et utiliser des collections éventuellement synchronisées

Contrainte

Toutes les variables d'instance et les variables de classe doivent être déclarées `private`.

Exercice 1 : Dessiner le billard

Objectifs spécifiques

- Créer et afficher une fenêtre (`JFrame`) contenant un panneau (`JPanel`)
- Manipuler la classe `Graphics`

Étape 1 : Classe Wall

Un billard est délimité par 4 parois (bords extérieurs du billard), chacune étant un objet de la classe `Wall`.

Créez la classe `Wall` qui contient une seule caractéristique : une référence (variable d'instance appelée `rectangle`) vers un objet de type `Rectangle`. *N.B. La classe* `Rectangle` *existe dans Java. Consultez sa documentation*.

Cet objet `Rectangle` permettra de dessiner une paroi du billard sous forme d'un rectangle.

Prévoyez dans la classe `Wall`, un constructeur à 4 arguments de type `int` : les coordonnées permettant de créer l'objet `rectangle` (cf. documentation). L'objet `rectangle` doit être créé et initialisé dans le constructeur.

Prévoyez également dans la classe `Wall` la méthode `draw(Graphics g)` qui ne retourne aucun résultat. Cette méthode doit dessiner sur l'objet `Graphics g` un rectangle dont les coordonnées sont reprises dans la variable `rectangle`. Pour ce faire, consultez les méthodes disponibles dans la classe `Graphics`.

> **Note**
>
> La méthode `drawRect(…)` dessine un rectangle vide, tandis que la méthode `fillRect(…)` dessine un rectangle plein.

Étape 2 : Classe Billiard

Créez la classe `Billiard` qui hérite de `JPanel`. Tout objet de la classe `Billiard` est caractérisé par un ensemble (liste) de parois verticales et un ensemble (liste) de parois horizontales : ces deux listes sont de type `ArrayList`. N'oubliez pas de les typer : elles sont de type `Wall` !

Ces deux listes sont des variables d'instance de la classe `Billiard` !

Créez de cette façon les 4 bords extérieurs de votre billard.

Vous avez donc placé, pour le moment, deux parois verticales et deux horizontales dans les listes.

Redéfinissez ensuite dans la classe `Billiard` la méthode : `public void paint (Graphics g).`

Cette méthode est appelée automatiquement lorsqu'un composant graphique doit être (re)dessiné. Elle est appelée notamment par le constructeur du composant graphique.

> **Note**
>
> Il est impératif de placer dans le corps de la méthode `paint` redéfinie un appel explicite à la méthode `paint` héritée (`super.paint(g)`).

Placez dans le code de cette méthode `paint` les instructions de dessin : appelez la méthode `draw` sur chacune des parois verticales et horizontales.

Étape 3 : Classe MainWindow

Créez une classe `MainWindow` qui hérite de la classe `JFrame`. Cette classe permettra d'afficher la fenêtre principale.

Faites-en sorte que soit affichée dans la fenêtre principale une occurrence du panneau `Billiard`. Utilisez le gestionnaire de tracé par défaut ; en effet, c'est la méthode `paint` du billard qui se chargera de dessiner le billard.

N'oubliez pas de gérer la fermeture de la fenêtre (gestion d'événement : si l'utilisateur clique sur l'icône X, l'application se termine) via `addWindowListener` !

```
public class MainWindow extends JFrame {
   private Billiard billiard;
   private Container mainContainer;

   public MainWindow () {

      // Appel au constructeur hérité en donnant un titre à la fenêtre
      super("Billiard");

      // Positionnement de la fenêtre à l'écran :
      setBounds(..., ..., ..., ...);

      // Gestion de la fermeture de la fenêtre si clic sur icône X :
      addWindowListener (new WindowAdapter() {
           public void windowClosing (WindowEvent e) {
              System.exit(0);
           }
      } );

      // Création d'une occurrence du panel représentant le billard
      billiard = new Billiard(...);

      // Récupération de la référence du conteneur de la fenêtre :
      mainContainer = this.getContentPane();

      /* Ajout du billard au conteneur de la fenêtre via le gestionnaire de tracé
         par défaut */
      mainContainer.add(billiard);

           ...

      //  Affichage de la fenêtre :
      setVisible(true);
   }
}
```

Créez une classe `Main` contenant la méthode `main` qui crée une occurrence de la classe `MainWindow`.

Si tout se passe bien, à l'exécution, vous devriez voir apparaître votre billard.

Exercice 2 : Déplacer une balle sur le billard

Objectif spécifique
- Créer un thread qui déplace une balle

Étape 1 : Dessiner une balle

Créez la classe `Ball`. Toute balle est caractérisée par :

- une référence (appelée `rectangle`) vers un objet de type `Rectangle` (la balle sera dessinée à l'intérieur des coordonnées de ce rectangle) ;
- une référence vers l'objet `Billiard` sur lequel apparaîtra la balle.

Prévoyez dans la classe `Ball` un constructeur à 5 arguments : une référence vers un objet de type `Billiard` et les 4 coordonnées de type `int` permettant de créer et initialiser le rectangle `rectangle`.

Prévoyez également dans la classe `Ball` la méthode `draw(Graphics g)` qui dessine la balle à l'intérieur des coordonnées du rectangle `rectangle`.

Adaptez la classe `Billiard` comme suit.

- Ajoutez une caractéristique supplémentaire au billard : une référence vers une balle.
- Créez et initialisez cette balle dans le constructeur de la classe `Billiard`.
- Adaptez la méthode `paint` du billard pour que la balle soit dessinée sur ce billard.

Étape 2 : Faire bouger la balle (sans rebondir sur les parois)

Principe

Un Thread, à intervalles réguliers, demande à la balle de se déplacer (via appel à la méthode `move` sur la balle). Pour bouger, la balle augmente les coordonnées x et y du rectangle dans lequel elle est dessinée, respectivement de `stepX` et `stepY`.

Adaptez la classe `Ball` comme suit.

- Ajoutez deux caractéristiques supplémentaires à la balle : `stepX` et `stepY`.
- Créez la méthode `move()` : augmentez les coordonnées X (+ `stepX`) et Y (+ `stepY`) du rectangle dans lequel est dessinée la balle.
- Créez la classe `MovementThread` qui hérite de `Thread` et qui contient une référence vers un billard. Le thread doit, à intervalle régulier, appeler la méthode `move()` sur la balle du billard .

Attention

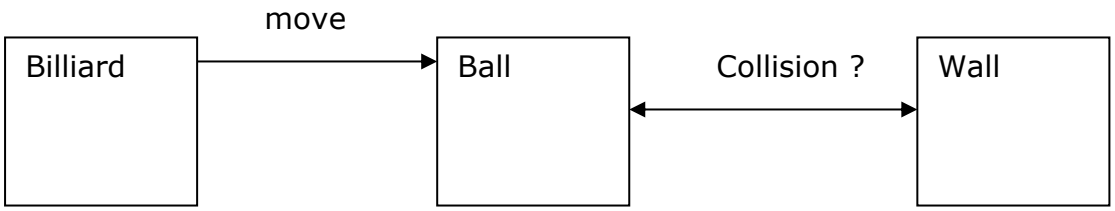
Le thread doit explicitement appeler la méthode `repaint()` sur l'objet de type `Billiard`, sous peine de ne pas voir le mouvement de la balle à l'écran. Il faut en effet demander de redessiner le contenu du `JPanel` après chaque appel de la méthode `move()` sur la balle. Ceci aura pour effet d'appeler la méthode `paint` de la classe `Billiard`.

Adaptez la classe `Billiard` : créez une occurrence du thread dans le constructeur et démarrez-le.

Étape 3 : Faire rebondir la balle sur les parois du billard

Principe

Chaque fois qu'on demande à la balle de bouger, elle interroge chacune des parois pour savoir si elle n'est pas entrée en collision avec elle. Si c'est le cas, elle doit modifier les coordonnées X et Y du rectangle dans lequel elle est dessinée pour simuler le rebondissement.



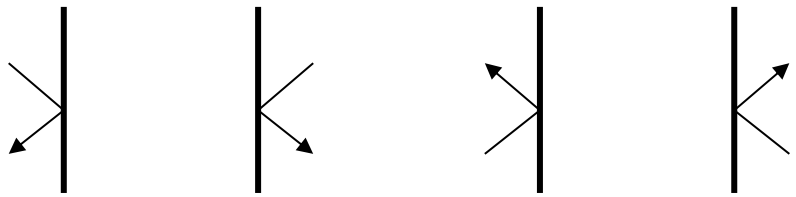
Adaptez la classe `Wall` : créez la méthode `collision` qui prend un argument de type `Ball` et qui retourne un booléen. Elle renvoie `true` si le rectangle dans lequel est dessinée la balle est en intersection avec le rectangle de la paroi ; elle renvoie `false` sinon.

Suggestion

Utilisez la méthode `intersects` de la classe `Rectangle` qui permet de détecter si deux rectangles sont en intersection.

Adaptez la méthode `move()` de la classe `Ball` comme suit.

- Bouclez sur les parois **verticales**.
- Sur chaque paroi, appelez la méthode collision.
- S'il y a collision, adaptez la nouvelle coordonnée **X** du rectangle dans lequel est dessinée la balle pour simuler le rebondissement (cf. ci-après).



- Faites de même sur les parois **horizontales** et, en cas de collision, adaptez la nouvelle coordonnée **Y** du rectangle de la balle pour simuler le rebondissement (cf. ci-après).

Exercice 3 : Transformer le billard en flipper : ajouter des obstacles et totaliser les points

Objectif spécifique

- Créer un thread qui affiche les points dans une autre fenêtre

Étape 1 : Créer des obstacles

Ajouter des parois supplémentaires **dans** le billard. Ces parois joueront le rôle d'obstacles pour la balle. La balle doit aussi rebondir sur ces obstacles.

Il suffit de créer des parois verticales ou horizontales (de faible épaisseur de préférence !) et de les placer dans la liste correspondante.

Suggestion

Une des solutions pour éviter que la balle ne "colle" à une paroi consiste à ajouter aux deux extrémités de chaque paroi verticale une paroi horizontale de la même largeur que la paroi verticale. Et inversement (c'est-à-dire ajouter aux deux extrémités de chaque paroi horizontale une paroi verticale de la même largeur que la paroi horizontale).

Étape 2 : Totaliser les points obtenus lorsque la balle rebondit sur les obstacles

Ajoutez une variable d'instance `points` dans la classe `Wall`. Mettez à jour le constructeur en conséquence.

Mettez à jour la création des parois dans la classe `Billiard` : attribuer 0 point aux parois extérieures du billard et un nombre de points non nul aux parois de type obstacle.

Ajoutez une variable d'instance `totalPoints` dans la classe `Billiard`.

Mettez à jour la méthode `move` de la classe `Ball` : à chaque collision de paroi, ajouter le nombre de points associés à la paroi au total des points du billard.

Étape 3 : Afficher le total des points ("en temps réel") dans une autre fenêtre

Créez la classe `CounterPanel` qui hérite de `JPanel`. La classe `CounterPanel` contient une référence vers le billard. Redéfinissez dans la classe `CounterPanel` la méthode `public void paint (Graphics g)`**.** Consultez les méthodes disponibles

dans la classe `Graphics` afin d'afficher sur l'objet `Graphics` le total des points du billard.

Créez une classe `PointsWindow` qui hérite de la classe `JFrame`. Faites-en sorte que soit affichée dans cette fenêtre une occurrence du panneau `CounterPanel`.

N'oubliez pas de gérer la fermeture de la fenêtre (gestion d'événement : si l'utilisateur clique sur l'icône X, l'application se termine) !

Suggestion

Mémoriser dans la classe `PointsWindow` une référence vers le billard.

Créez une occurrence de `PointsWindow` dans le constructeur de la classe `MainWindow`.

Créez la classe `PointsCountingThread` qui hérite de `Thread` et qui contient une référence vers un panneau de type `CounterPanel`. Le thread doit, à intervalles réguliers, rafraîchir le contenu du panneau de type `CounterPanel` afin d'afficher "en temps réel" le total des points obtenus.

Créez une occurrence du thread dans le constructeur de la classe `CounterPanel` et démarrez-le.

Exercice 4 : Changer la couleur des balles

Objectif spécifique

- Créer un thread qui change la couleur de la balle

Étape 1 : Associer une couleur à une balle

Adaptez la classe `Ball` : ajouter la couleur comme caractéristique d'une balle. Utilisez la classe `Color` (cf. documentation). Par défaut, toute balle est noire à sa création.

Adaptez la méthode `draw` de la classe `Ball` : consultez la documentation de la classe `Graphics` afin de colorer la balle.

Ajoutez la méthode `changeColor` dans la classe `Ball` : cette méthode doit permettre de changer la couleur de la balle.

Suggestion

Prévoyez un tableau de couleurs dans la classe `Ball` (attention : un seul tableau quel que soit le nombre de balles créées !). Bouclez sur les couleurs du tableau pour changer de couleur. Dès que vous arrivez à la fin du tableau, repositionnez-vous au début du tableau.

Étape 2 : Créer le thread qui change la couleur

Créez la classe `ColorChangingThread` qui hérite de `Thread` et qui contient une référence vers le panneau `billiard`. Le thread doit, à intervalles réguliers, changer la couleur de la balle du billard.

Adaptez la classe `Billiard` : créez une occurrence du thread dans le constructeur et démarrez-le.

Exercice 5 : Lancer plusieurs balles sur le billard/flipper

Objectifs spécifiques
- Gérer les événements liés à un bouton "Start" générant plusieurs balles
- Utiliser une collection synchronisée

Étape 1 : Stocker plusieurs balles dans une liste synchronisée

Les balles doivent **impérativement être stockées dans une liste synchronisée**.

Note

L'utilisation d'une liste synchronisée évite les conflits d'accès éventuels lorsque plusieurs threads y accèdent !

N'oubliez pas de typer cette liste : elle est de type `Ball` !

Liste synchronisée

```
import java.util.Collections;

…

ArrayList<Ball> ballsArray = new ArrayList<>();

List<Ball> balls;

…

balls = Collections.synchronizedList(ballsArray);

…

balls.add(new Ball());
```

Adaptez la classe `Billiard`. Remplacez la référence vers une balle par une liste **synchronisée** de balles. Cette liste est vide à la création du billard.

Modifiez la méthode `paint(Graphics g)` de la classe `Billiard` : bouclez sur la liste de balles pour dessiner toutes les balles de la liste.

Adaptez les classes `ColorChangingThread` et `MovementThread` afin de boucler sur les balles de la liste de balles.

Étape 2 : Ajouter un bouton "start" générant les balles

Adaptez la classe `MainWindow`. Créez-y un second `JPanel` contenant un `JButton` libellé "Start". Placez le panneau `billiard` au ***CENTRE*** de la fenêtre et le panneau contenant le bouton au ***SUD***.

Associez une **gestion d'événement** au bouton `Start` : à chaque clic sur ce bouton, une balle est générée dans la liste des balles du billard.

Étape 3 : Faire rebondir les balles les unes contre les autres

Adaptez la méthode `move` de la classe `Ball` : faites-en sorte que les balles qui entrent en collision les unes avec les autres, rebondissent ou, plus exactement, repartent dans des directions opposées.

> **Suggestion**
>
> Inverser simplement le sens des `stepX` et `stepY` en cas de collision entre balles.

Exercice 6 : Faire disparaître des balles dans des pièges

Objectif spécifique
- Supprimer des éléments d'une liste synchronisée

Étape 1 : Créer des pièges

Créez la classe `Hole` afin de gérer les pièges, à savoir des trous dans lesquels tombent et disparaissent les balles. Tout piège est caractérisé par une référence (variable d'instance appelée `rectangle`) vers un objet de type `Rectangle` (le piège sera dessiné à l'intérieur des coordonnées de ce rectangle).

Prévoyez dans la classe `Hole` la méthode `draw(Graphics g)` qui dessine le piège à l'intérieur des coordonnées du rectangle `rectangle`. Faites-en sorte qu'un piège soit dessiné sous forme d'un cercle plein de couleur noire (cf. documentation de la classe `Graphics`).

Adaptez la classe `Billiard`. Ajoutez-y une liste de pièges. Créez et initialisez des pièges dans le constructeur de la classe `Billiard`. Modifiez la méthode `paint(Graphics g)` de la classe `Billiard` afin que soient dessinés tous les pièges.

Étape 2 : Tester si les balles tombent dans les pièges

Adaptez la classe `Hole`.

Créez la méthode `disappearance` qui prend un argument de type `Ball` et qui retourne un booléen. Elle renvoie `true` si le rectangle dans lequel est dessiné le piège est en intersection avec le rectangle dans lequel est dessinée la balle ; elle renvoie `false` sinon.

Adaptez la classe `Ball`.

- Ajoutez une caractéristique supplémentaire à la balle : un **booléen** appelé
- `toEliminate`. Ce booléen doit être initialisé à `false` à la création de la balle.
- Adaptez la méthode `move()` de la classe `Ball` : sur chaque piège, appelez la méthode `disappearance` ; si elle retourne `true`, notez que la balle est à supprimer en plaçant `true` dans le booléen `toEliminate`.

Étape 3 : Faire disparaître du billard les balles tombées dans les pièges

Adaptez la classe `MovementThread`.

Prévoyez deux boucles distinctes sur la liste des balles :

- première boucle pour appeler la méthode `move` sur chaque balle ;
- seconde boucle pour supprimer de la liste les balles qui sont notées à supprimer.