TP2
Les barrières

Le point de départ de ce TP est la version de BallWorld obtenue à la fin du TP 1.

Exercice 1: Stopper les balles

Vous devez maintenant modifier le programme pour arriver au comportement suivant: quand une balle, au cours de ses mouvements, se trouve dans la zone diagonale du monde (là où la différence entre `xpos` et `ypos` en valeur absolue est inférieure à 2), elle arrête de se déplacer.

Notez qu'une balle peut sauter par-dessus la zone diagonale d'un seul mouvement, sans qu'elle s'arrête.

Quand toutes les balles sont arrêtées à la diagonale, elles doivent toutes se réveiller et continuer jusqu'à ce qu'ils s'arrêtent à nouveau sur la diagonale. Ce comportement se poursuit à l'infini.

Indication :

Cette forme de synchronisation s'appelle une *barrière* à N threads (N est le nombre de balles, 4 dans notre cas). Elle peut être réalisée en utilisant 2 sémaphores, que nous allons appeler `barrierSem` et `barrierKeeperSem`. Chaque *thread* qui arrive à la barrière dépose un jeton dans `barrierKeeperSem` puis essaye de récupérer un jeton de `barrierSem`, ce qui va le mettre en attente.

Un autre *thread*, que nous appellerons `BarrierKeeper`, gère la barrière en attendant N fois sur le sémaphore `barrierKeeperSem` (les jetons déposés par chacun des *threads* bloqués devant la barrière), puis en déposant N jetons dans `barrierSem`, ce qui libérera les *threads* bloqués devant la barrière.

Remarques :

- La classe sémaphore à utiliser est : `java.util.concurrent.Semaphore.`
- Si l'on commence à tuer les balles (TP1, exo2) alors que certaines sont encore bloquées, votre programme doit permettre de tuer les balles bloquées aussi. Ils est acceptable alors que les balles soient débloquées avant d'être tuées.

Question : Est-ce que la barrière fonctionne comme prévu si l'on oublie de déclarer les objets `java.util.concurrent.Semaphore` comme étant *FIFO* (dans le constructeur) ? Expliquer pourquoi.

Exercice 2 : Utilisation de CyclicBarrier

Le paquetage `java.util.concurrent` inclut la classe `CyclicBarrier`, qui fournit un moyen plus commode d'implémenter des barrières de synchronisation.

Réécrire le programme de l'exercice précédent en utilisant cette classe au lieu de sémaphores.

Exercice 3 : Votre propre CyclicBarrier

Passons maintenant à un exercice plus difficile: implémenter votre propre classe `MyCyclicBarrier` qui offre des fonctionnalités similaires à celles de la classe Java du même nom. On se contentera d'une version plus simple avec les spécifications suivantes:

```
public class MyCyclicBarrier {
     public MyCyclicBarrier(int parties);
     public void awaitEverybody();
     public void reset();
}
```

L'implémentation se fera en utilisant exclusivement les fonctions de moniteur des objets Java (`wait`, `notify`, `notifyAll`).

Implémenter la classe et ensuite l'utiliser pour résoudre l'exercice 2 à nouveau.