Quelques exercices autour des barrières

Dans ces exercices, vous vous entraînerez à créer des threads et à utiliser des sémaphores pour synchroniser leur comportement. Nous partirons d'un programme qui ouvre une fenêtre dans laquelle quatre "balles" se mettent en mouvement et rebondissent contre les murs. Ce programme contient 3 classes :

- `Ball.java`

  Une instance de cette classe est un ballon, qui vit dans un BallWorld. La classe est une sous-classe de Thread et sa méthode run est une boucle infinie où la balle met régulièrement à jour sa position, appelle le monde à repeindre et dort pendant un court instant. La balle a également la capacité de se dessiner elle-même, dans un contexte graphique. De plus, lors de sa création (c'est-à-dire dans son constructeur), la balle s'inscrit dans le monde dans lequel elle vit en appelant la méthode addBall du monde.

- `BallWorld.java`

  Une instance de Ballword représente le monde qui contient différentes balles stockées dans un ArrayList. Un monde est une sous-classe de JPanel, la classe Swing utilisée comme surface de dessin. Par conséquent, le monde peut se dessiner lui-même, ce qu'il fait dans la méthode PaintComponent en demandant à toutes les billes qu'il contient de se dessiner.

- `Balls.java`

  Cette classe contient la méthode principale, qui consiste à créer un monde et quelques balles.

Vous noterez que le flux de contrôle dans cet exemple est plutôt subtil. Le déplacement des balles et leur représentation dans le monde (via repaint) sont initiés indépendamment par chacune des billes, car chacune d'entre elles s'exécute dans un flot d'exécution distinct.

De ce fait, différentes boules peuvent exécuter doMove() et world.repaint() simultanément. Appeler world.repaint() en même temps est acceptable, comme indiqué dans la documentation de Swing. Mais l'appel de repaint() déclenche également l'appel de la méthode draw() de toutes les balles enregistrées dans le monde. Là encore, c'est normal, car les deux méthodes draw() et doMove() sont synchronisées, ce qui signifie qu'elles ne seront pas appelées simultanément pour un objet donné.

Comme vous pouvez le voir, ce programme très simple a déjà une complexité certaine. Il est acceptable d'avoir une telle conception pour des programmes simples, mais pour tout programme plus important, son comportement concurrent doit être conçu de manière structurée, sinon il sera extrêmement compliqué.

Regardez les classes et assurez-vous que vous comprenez le code. Compiler et exécuter le programme.

Exercise

Vous devez modifier le programme pour obtenir le comportement suivant : Lorsqu'une balle se trouve après un de ses mouvements dans la zone diagonale du monde (c'est-à-dire où x est très proche de y), elle se "fige", c'est-à-dire qu'elle cesse de bouger. Lorsque toutes les balles se sont figées, elles se réveillent toutes et continuent à rebondir jusqu'à ce qu'elles figent à nouveau en diagonale. Cette alternance de comportement se répète à l'infini.

Attention : une balle peut sauter par-dessus la diagonale en un seul coup ; elle ne se fige pas pour autant.

Si vous avez étudiez le code avec attention vous avez remarqué qu'une balle se fige un court instant lorsqu'elle est dans la diagonale.

A special barrier synchronization process is also needed, which repeatedly acquire the barrier semaphore N times, followed by releasing all the continue semaphores.

Etape 1 : utilisez la CyclicBarrier du paquet java.util.concurrent

Etape 2 : implémentez votre propre CyclicBarrier en utilisant les sémaphores

Etape 3 : implémentez votre propre CyclicBarrier en utilisant les moniteurs