TP de Compléments en Programmation Orientée Objet n° 9-10 : *Multithreading* (primitives de synchronisation)

I) Synchronisation et moniteurs

Exercice 1 : Compteurs

On considère la classe `Compteur`, que nous voulons tester et améliorer :

```
1  public class Compteur {
2      private int compte = 0;
3      public int getCompte() { return compte; }
4      public void incrementer() { compte++; }
5      public void decrementer() { compte--; }
6  }
```

1. À cet effet, on se donne la classe `CompteurTest` ci-dessous :

```
1  public class CompteurTest {
2      private final Compteur compteur = new Compteur();
3
4      public void incrementerTest() {
5          compteur.incrementer();
6          System.out.println(compteur.getCompte() + " obtenu après incrémentation");
7      }
8
9      public void decrementerTest() {
10         compteur.decrementer();
11         System.out.println(compteur.getCompte() + " obtenu après décrémentation");
12     }
13 }
```

   Écrivez un `main` qui lance sur une seule et même instance de la classe `CompteurTest` des appels à `incrementerTest` et `decrementerTest` depuis des *threads* différents. Pour vous entrainer à utiliser plusieurs syntaxes, lancez en parallèle :
   — une décrémentation à partir d'une classe locale, dérivée de `Thread` ;
   — une décrémentation à partir d'une implémentation anonyme de `Runnable` ;
   — une incrémentation à partir d'une lambda-expression obtenue par lambda-abstraction (syntaxe `args -> result`) ;
   — une incrémentation à partir d'une lambda-expression obtenue par référence de méthode (syntaxe `context::methodName`).

2. On souhaite maintenant qu'il soit garanti, même dans un contexte *multi-thread*, que la valeur de `compte` (telle que retournée par `getCompte`) soit toujours égale au nombre d'exécutions d'`incrementer` moins le nombre d'exécutions de `decrementer` ayant terminé avant le retour de `getCompte` (rappel : l'incrémentation `compte++` et la décrémentation `compte--` ne sont pas des opérations atomiques).
   Obtenez cette garantie en ajoutant le mot-clé **`synchronized`** aux endroits adéquats dans la classe `Compteur`.

3. Est-ce que les modifications de la question précédente assurent que `incrementerTest` et `decrementerTest` affichent bien la valeur du compteur obtenue après, respectivement, l'appel à `incrementer` ou à `decrementer` fait dans chacunes des deux méthodes de test ? Comment modifier `CompteurTest` pour que ce soit bien le cas ?

4. On veut ajouter à la classe `Compteur` la propriété supplémentaire suivante : « `compte` n'est jamais être négatif ». Celle-ci peut être obtenue en rendant l'appel à `decrementer` bloquant quand `compte` n'est pas strictement positif. Modifiez la classe `Compteur` en introduisant les `wait()` et `notify()` nécessaires dans la classe `Compteur`.

Exercice 2 : Un peu plus loin que les moniteurs

Le problème lecteurs-rédacteur est un problème d'accès à une ressource devant être partagée par deux types de processus :

— les lecteurs, qui consultent la ressource sans la modifier,
— les rédacteurs, qui y accèdent pour la modifier.

Pour que tout se passe bien, il faut que, lorsqu'un rédacteur a la main sur la resource, aucun autre processus n'y accède « simultanément » [1]. En revanche, on ne veut pas interdire l'accès à plusieurs lecteurs simultanés.

Malheureusement, les moniteurs de Java ne gèrent directement que l'exclusion mutuelle [2]. Pour implémenter le schéma lecteurs-rédacteur, il faut donc une classe dédiée.

Nous allons procéder en trois étapes :

— définition d'une classe verrou,
— association d'un verrou et d'une ressource,
— mise en place d'un test de lectures écritures concurrentes.

Il est probable que vous oublierez des choses au départ. Vous y reviendrez et procéderez aux ajustements au moment des tests. Vous trouverez également quelques conseils en fin d'exercice.

1. Définissez une classe, dont les objets seront utilisés comme des verrous, nous l'appellerons `ReadWriteLock`. Ils contiennent :
   — un booléen pour dire si un écrivain est actuellement autorisé ;
   — le nombre de lecteurs actuellement actifs sur la ressource ;
   — la méthode `dropReaderPrivilege()` qui décrémente le nombre de lecteurs actuels ;
   — la méthode `dropWriterPrivilege()` qui libère la ressource de son rédacteur ;
   — les méthodes `acquireReaderPrivilege()` et `acquireWriterPrivilege()`, bloquantes sur le moniteur du verrou, pour demander un droit d'accès en lecture ou en écriture.

   Testez cette classe. Par exemple, la séquence suivante ne doit pas bloquer :

```
1   val lock = new ReadWriteLock(); lock.acquireReaderPrivilege(); lock.acquireReaderPrivilege();
```

   mais celle-ci, oui :

```
1   val lock = new ReadWriteLock(); lock.acquireReaderPrivilege(); lock.acquireWriterPrivilege();
```

   (On peut la débloquer en appelant `lock.dropReaderPrivilege()` dans un autre thread.) et celle-là aussi :

```
1   val lock = new ReadWriteLock(); lock.acquireWriterPrivilege(); lock.acquireReaderPrivilege();
```

   (On peut la débloquer en appelant `lock.dropWriterPrivilege()` dans un autre thread.)

2. Écrire une classe `ThreadSafeReadWriteBox`, encapsulant une ressource de type `String` et une instance de verrou `ReadWriteLock`. Utilisez le verrou dans le getteur et le setteur de la ressource, afin de garder les accès en lecture et écriture (en acquérant le privilège pertinent avant l'accès ; puis en le libérant après l'accès).

1. On évite ainsi de créer des accès conflictuels non synchronisés, i.e. des accès en compétition.
2. Le moniteur n'appartient qu'à un seul *thread* en même temps, à l'exclusion de tout autre.

3. Ecrivez une classe de test dont le `main()` manipule une instance de `ThreadSafeReadWriteBox` contenant la chaîne "`Init`". Vous lancerez deux threads changeant la valeur de la ressource en "`A`" et "`B`" respectivement, et 10 autres threads qui se contenteront d'afficher la ressource. On aura donc 2 opérations d'écritures et 10 de lectures.

   Pour se rendre compte de l'ordonnancement et de la concurrence, modifiez la méthode `set` de `ThreadSafeReadWriteBox` pour qu'elle attende une seconde avant d'écrire.

   Modifiez également `get` pour qu'elle attende aléatoirement entre 0 et deux secondes.

   Etudiez les ordonnancements possibles des lectures et écritures et donnez une estimation du temps attendu. Vérifiez bien que votre test s'exécute dans ces délais.

4. `ReadWriteLock` (comme les verrous explicites fournis par le package `java.util.concurrent.locks` du JDK) a un défaut majeur par rapport aux moniteurs : rien n'oblige à libérer un verrou après son acquisition (pour les moniteurs, c'était le cas car l'acquisition se fait en entrant dans le bloc **`synchronized`** et la libération en en sortant). Un tel oubli provoquerait typiquement un *deadlock*.

   L'API de la classe `ThreadSafeReadWriteBox` qui encapsule un `ReadWriteLock`, est API plus sûre car il est impossible pour l'utilisateur d'oublier de libérer le verrou encapsulé (c'est géré par le getteur et le setteur). Mais cette classe est trop spécialisée (seulement lecture et affectation d'un `String`).

   Pourriez-vous proposer une nouvelle interface pour `ReadWriteLock` qui n'ait pas ce problème ? (pensez fonctions d'ordre supérieur ou bien alors, documentez vous sur les blocs *try-with-resource* et l'interface `Autocloseable`)

   Écrivez une classe implémentant cette API en se basant sur une instance encapsulée (privée) de `ReadWriteLock`.

Quelques conseils :

— On rappelle que pour utiliser et libérer une ressource (ici le verrou) la bonne façon de faire est de la forme `acquerir(R); try { instructions } finally { liberer(R); }` ainsi même s'il y a un **`return`** dans les instructions, la ressource est libérée.

— Pensez à distinguer `notify` et `notifyAll`, n'en ajoutez pas non plus partout. Justifiez bien leur écriture en vous demandant qui peut être en état d'attente.

II) Accès en compétition et *thread-safety*

Exercice 3 : Accès en compétition

Les classes suivantes interdisent-elles les accès en compétition au contenu de leurs instances ?

Attention : pour cet exercice, on considère que le « contenu », c'est aussi bien les attributs que les attributs des attributs, et ainsi de suite.

Rappel : 2 accès à une même variable partagée sont en compétition si au moins l'un est en écriture et il n'y a pas de relation arrivé-avant entre les deux accès.

```
1  public final class Ressource {
2      public static class Data { public int x; }
3      public final Data content;
4      public Ressource2(int x) {
5          content = new Data();
6          content.x = x;
7      }
8  }
9
```

```java
public final class Ressource2 {
    private String content;
    private boolean pris = false;

    private synchronized void lock() throws InterruptedException {
        while(pris) wait();
        pris = true;
    }

    private synchronized void unlock() {
        pris = false;
        notify();
    }

    public void set(String s) throws InterruptedException {
        lock();
        try { content = s; }
        finally { unlock(); }
    }

    public String get() throws InterruptedException {
        lock();
        try { return content; }
        finally { unlock(); }
    }
}

public final class Ressource3 {
    public static class Data {
        public final int x;
        public Data(int x) { this.x = x; }
    }

    public volatile Data content;

    public Ressource3(int x) { content = new Data(x); }
}
```

Exercice 4 : *Thread-safe* ?

Une classe est *thread-safe* si sa spécification reste vraie dans un contexte d'utilisation multi-thread. Quelles classes parmi les suivantes sont-elles *thread-safe* pour la spécification : « à tout moment, la valeur retournée par le getteur est égale au nombre d'appels à `incremente` déjà entièrement exécutés » ?

```java
public final class Compteur {
    private int i=0;
    public synchronized void incremente() { i++; }
    public synchronized int get() { return i; }
}

public final class Compteur2 {
    private volatile int i = 0;
    public void incremente() { i++; }
    public int get() { return i; }
}

public final class Compteur3 {
    private int i=0;
    public synchronized void incremente() { i++; }
    public int get() { return i; }
}
```