TP de Compléments en Programmation Orientée Objet 9 : *Multithreading* (primitives de synchronisation)

A finir, si ce n'est déjà fait: TP7, Exercice 7 et TP8, 2 premiers exercices.

I) Accès en compétition et *thread-safety*

Exercice 1 : 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 Ressource(int x) {
5          content = new Data();
6          content.x = x;
7      }
8  }
9
10 public final class Ressource2 {
11     private String content;
12     private boolean pris = false;
13
14     private synchronized void lock() throws InterruptedException {
15         while(pris) wait();
16         pris = true;
17     }
18
19     private synchronized void unlock() {
20         pris = false;
21         notify();
22     }
23
24     public void set(String s) throws InterruptedException {
25         lock();
26         try { content = s; }
27         finally { unlock(); }
28     }
29
30     public String get() throws InterruptedException {
31         lock();
32         try { return content; }
33         finally { unlock(); }
34     }
35 }
36
37 public final class Ressource3 {
38     public static class Data {
39         public final int x;
40         public Data(int x) { this.x = x; }
41     }
42     public volatile Data content;
43
44     public Ressource3(int x) { content = new Data(x); }
45 }
```

Exercice 2 : *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 *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 » ?

```
1  public final class Compteur {
2      private int i=0;
3      public synchronized void incremente() { i++; }
4      public synchronized int get() { return i; }
5  }
6
7  public final class Compteur2 {
8      private volatile int i = 0;
9      public void incremente() { i++; }
10     public int get() { return i; }
11 }
12
13 public final class Compteur3 {
14     private int i=0;
15     public synchronized void incremente() { i++; }
16     public int get() { return i; }
17 }
```

II) Synchronisation et moniteurs

Exercice 3 : 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 entraîner 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 `CompteurTest` en introduisant les `wait()` et `notify()` nécessaires.