TP n° 9 : *Multithreading* (primitives de synchronisation)
(Correction)

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.

```java
public final class Ressource {
   public static class Data { public int x; }
   public final Data content;
   public Ressource(int x) {
      content = new Data();
      content.x = x;
   }
}

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); }
}
```

Correction :

1. `Ressource` n'empêche pas les accès en compétition : en effet, si on crée `Ressource r = new Ressource();` alors l'accès à l'attribut `r.content.x`, possible en lecture comme en écriture, n'est protégé par aucun mécanisme de synchronisation. Par exemple, il peut y avoir des problèmes de non-synchronisation des caches.
2. `Ressource2` est inutilement compliquée... mais assure en effet bien l'absence d'accès en compétition.

   Explication : les attributs sont privés et uniquement accessibles, respectivement en lecture et en écriture, via des méthodes dédiées.

   `pris` est uniquement accédé via les méthodes synchronisées `lock` et `unlock`, donc il n'y a pas d'accès en compétition à cet attribut.

   Quant à `content`, il est accédé via les méthodes publiques `get` et `set`. Or, l'appel à une de ces méthodes consiste en 3 actions : `lock`, accès, `unlock`, ordonnées par l'ordre du programme, donc par la relation arrivé-avant. De plus les méthodes `lock` et `unlock` décrivent un mécanisme d'acquisition/libération de ressource (symbolisée par le booléen `pris`) garantissant qu'entre 2 exécutions de `lock`, il y a nécessairement eu une exécution de `unlock` (le tout ordonné par arrivé-avant, grâce à **`synchronized`**).

   Bout-à-bout, cela signifie que tout accès à `content` (dans appel à `get` ou `set`) arrive-avant tout autre accès qui serait exécuté chronologiquement après. Il n'y a donc pas d'accès en compétition à cet attribut non plus.
3. `Ressource3` ne garantit pas l'absence d'accès en compétition !

   Les apparences sont trompeuses : tous les attributs déclarés dans `Ressource3` et sa classe imbriquée sont **`final`** ou **`volatile`**, ce qui garantit en effet que l'accès à ceux-ci se fait sans compétition.

   Mais, cela ne veut pas dire qu'on ne peut pas avoir d'accès en compétition via cette classe, en « ajoutant » des attributs par extension. Exemple :

```
1  Ressource3 r = new Ressource3(12);
2  class Data2 extends Ressource3.Data { // c'est ici que ça se passe !
3     int y = 0;
4     data2(int x) {
5        super(x);
6        y = x;
7     }
8  }
9  r.content = new Data2(13);
10
11 // à partir de là , il est possible d'accéder, en compétition, à ((Data2)
       r.content).y
```

   Pour être tranquille, il faut empêcher l'extension de `Ressource3.Data` en ajoutant **`final`** par exemple.

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 {
```

```
   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; }
}
```

Correction : `Compteur` est *thread-safe*, car **`synchronized`** force à finir l'exécution de `incremente` avant de commencer `get` ; par ailleurs, cela ajoute une relation arrivé-avant entre les accès, ce qui rend le changement visible.
`Compteur2` ne l'est pas car `i++` n'est pas une opération atomique. En effet : si `incremente` est exécutée par plusieurs *threads* en même temps, on a vu dans le cours qu'il était possible qu'une incrémentation se retrouve « oubliée ».
`Compteur3` ne l'est pas, car le **`synchronized`** manquant fait que la relation arrivé-avant entre l'accès en lecture de `get` et celui en écriture de `incremente` n'existe plus. C'est un accès en compétition, rien ne force plus les caches à être synchronisés. Il est possible que `get` ne voie pas le dernier `incremente` même si celui-ci a fini d'être exécuté.

II) Synchronisation et moniteurs

Exercice 3 : Compteurs

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

```
public class Compteur {
   private int compte = 0;
   public int getCompte() { return compte; }
   public void incrementer() { compte++; }
   public void decrementer() { compte--; }
}
```

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

```
public class CompteurTest {
   private final Compteur compteur = new Compteur();

   public void incrementerTest() {
      compteur.incrementer();
      System.out.println(compteur.getCompte() + " obtenu après incrémentation");
   }

   public void decrementerTest() {
      compteur.decrementer();
      System.out.println(compteur.getCompte() + " obtenu après décrémentation");
   }
}
```

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`).

Correction : Dans le `main` (ou votre méthode de test) :

```
CompteurTest compteur = new CompteurTest();

class MyThread extends Thread{
   public void run() {
      compteur.decrementerTest();
   }
 }
 MyThread t1 = new MyThread();

 Runnable r = new Runnable(){
      public void run() {
        compteur.decrementerTest();
      }
    };
 Thread t2 = new Thread(r);

 Thread t3 = new Thread(() -> compteur.incrementerTest());
 Thread t4 = new Thread( compteur::incrementerTest);

 //mettre tous les start en même temps augmente les chances
 // d'avoir des comportements différents suivant les exécutions.
 t1.start();
 t2.start();
 t3.start();
 t4.start();

 t1.join(); //facultatif
 t2.join();
 t3.join();
 t4.join();
```

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`.

Correction :

```
public class Compteur {
   private int compte = 0;
   public synchronized int getCompte() { return compte; }
   public synchronized void incrementer() { compte++; }
   public synchronized void decrementer() { compte--; }
}
```

On ajoute **`synchronized`** à `incrementer` et `decrementer` afin de les rendre ato-

miques par rapport au `Compteur`, ce qui les empêche leurs exécutions de s'entrelacer et de rendre la valeur de `compteur` incohérente.
Par ailleurs, pour être sûr que `getCompte` ne retourne pas une valeur intermédiaire incohérente, on déclare aussi cette méthode **`synchronized`** (elle ne s'exécutera jamais en même temps qu'une incrémentation ou décrémentation).

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 ?

   Correction : Non, ce n'est pas garanti car en exécutant plusieurs fois `incrementerTest` et `decrementerTest` les appels à `decrementer` et `incrementer` et les affichages s'entrelacent sans synchronisation (il peut donc y avoir plusieurs incrémentations, par exemple, entre deux affichages).
   Ajouter **`synchronized`** aux méthodes de `CompteurTest` suffit à régler ce problème (elles s'exécuteront en exclusion mutuelle l'une de l'autre, la séquence incr/décrémentation + affichage devenant atomique).
   L'instance de `compteur` encapsulée n'étant pas partagée avec un autre objet, il n'y a pas besoin d'une synchronisation commune, donc synchroniser sur **`this`** (l'instance courante de `CompteurTest`) fonctionne bien.

```java
public class CompteurTest {
   private final Compteur compteur = new Compteur();

   public synchronized void incrementerTest() {
      compteur.incrementer();
      System.out.println(compteur.getCompte() + " obtenu après
          incrémentation");
   }

   public synchronized void decrementerTest() {
      compteur.decrementer();
      System.out.println(compteur.getCompte() + " obtenu après
          décrémentation");
   }
}
```

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.

   Correction :

```java
public class CompteurTest {
   private final Compteur compteur = new Compteur();

   public synchronized void incrementerTest() {
      compteur.incrementer();
      System.out.println(compteur.getCompte() + " obtenu après
          incrémentation");
      notify();
   }

   public synchronized void decrementerTest(){
```

```
11          while(compteur.getCompte() <1){
12             try{
13                wait();
14             }
15             catch(InterruptedException e){
16             }
17          }
18          compteur.decrementer();
19          System.out.println(compteur.getCompte() + " obtenu après
                décrémentation");
20       }
21    }
```

Cette nouvelle classe `CompteurTest` est en réalité proche de ce qu'on attend du mécanisme appelé « sémaphore » (regardez `java.util.concurrent.Semaphore`), servant à modéliser une resource disponible en nombre fini et pour laquelle il faut gérer un certain nombre de « permis » (pour comparer avec `CompteurTest` : l'initialisation ne se fait pas à zéro, mais au nombre total de permis).
Remarque : la méthode `decrementer`, à cause du **`throws`**, nécessite d'insérer dans `decrementerTest` le bloc **`try/catch`** qui va bien :
La classc `Compteur` n'est pas modifiée.