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

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`).

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

```
1  var compteur = new CompteurTest();
2
3  class maClasse extends Thread {
4      public void run(){compteur.decrementerTest();}
5  }
6  new Thread(new maClasse()).start();
7
8  new Thread(new Runnable(){
9      public void run(){compteur.decrementerTest();}
10 }).start();
11
12 new Thread( () -> compteur.incrementerTest() ).start();
13
```

```
14  new Thread(compteur::incrementerTest).start();
```

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 :

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

   On ajoute **`synchronized`** à `incrementer` et `decrementer` afin de les rendre atomiques 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'exécuter 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.

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

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`.

Correction :

```
1  public class Compteur {
2      private int compte = 0;
3
4      public synchronized int getCompte() { return compte; }
5
6      public synchronized void incrementer() {
7          compte++;
8          notify();
9      }
10
11     // Propage InterruptedException si wait() est interrompu.
12     public synchronized void decrementer() throws InterruptedException {
13         while (compte <= 0) wait();
14         compte--;
15     }
16 }
```

Cette nouvelle classe `Compteur` 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 `Compteur` : 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 :

```
1  public class CompteurTest {
2      private final Compteur compteur = new Compteur();
3
4      public synchronized void incrementerTest() {
5          compteur.incrementer();
6          System.out.println(compteur.getCompte() + " obtenu après incrémentation");
7      }
8
9      public synchronized void decrementerTest() {
10         try {
11             compteur.decrementer();
12             System.out.println(compteur.getCompte() + " obtenu après décrémentation");
13         } catch (InterruptedException e) {
14             System.out.println("Erreur : exécution du thread interrompue pendant
                   l'attente.");
15         }
16     }
17 }
```

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.)

   Correction :

```
1  public class ReadWriteLock {
2      private boolean pris = false;
3      private int nbLecteur = 0;
4
5      public synchronized void dropReaderPrivilege() {
6          if (nbLecteur > 0) nbLecteur--; // il faut notifier, car qq'un peu attendre
7          // on devrait lever une exception pour un nb <0 "Mauvaise utilisation"
8          notify(); // simple suffit
9      }
10
11     public synchronized void dropWriterPrivilege() {
12         pris = false; // il faut notifier
13         notifyAll(); // penser à notifyAll, avec notify un seul lecteur qui attend serait
               débloqué
14     }
15
16     // bloquant jusqu'à  l'obtention du privilège
```

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.

```
17      public synchronized void acquireReaderPrivilege() throws InterruptedException {
18          while (pris) wait();
19          nbLecteur++;
20          // notify(); ne servirait à rien
21      }
22
23      // bloquant jusqu'à  l'obtention du privilège
24      public synchronized void acquireWriterPrivilege() throws InterruptedException {
25          while (pris || nbLecteur > 0) wait();
26          pris = true;// notify(); ne servirait à rien
27      }
28
29  }
```

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).

Correction :

```
1   public class ThreadSafeReadWriteBox {
2       private final ReadWriteLock lock;
3       private String content;
4
5       public ThreadSafeReadWriteBox(String x) {
6           content = x;
7           lock = new ReadWriteLock();
8       }
9
10      private static void attendExact(int x) {
11          try {
12              Thread.sleep(x);
13          } catch (InterruptedException ex) {
14              System.out.println("InterruptedException non traitée");
15          }
16      }
17
18      private static void attendJusquA(int x) {
19          attendExact((int) (x * Math.random()));
20      }
21
22      public void set(String v) {
23          try {
24              lock.acquireWriterPrivilege();
25              attendExact(1000); // attendre 1 secondes
26              content = v;
27          } catch (InterruptedException ex) {
28              System.out.println("InterruptedException non traitée");
29          } finally { // non, probablement que ce n'est pas dans un finally, en cas
                        d'interruption, ca ferait même une libération de trop (mais dans le monde des
                        booléans ca n'a pas d'importance)
30              lock.dropWriterPrivilege();
31          }
32      }
33
34      public String get() throws InterruptedException {
35          lock.acquireReaderPrivilege();
36          try {
37              attendJusquA(2000); // attendre jusqu'à  2 secondes
38              return content;
39          } finally {
40              lock.dropWriterPrivilege();
41          } // le finally est nécessaire à cause de l'ordre return, puis libération
```

```
42      }
43  }
```

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.

Correction :

```
1  public class TestReadWriteLock {
2      public static void main(String[] args) {
3          ThreadSafeReadWriteBox x = new ThreadSafeReadWriteBox("Init");
4
5          new monThreadWrite(x, "A").start(); // remplace Init par A;
6          new monThreadWrite(x, "B").start(); // remplace Init par B;
7
8          for (int i = 0; i < 10; i++) new monThreadRead(x).start(); // effectue une lecture
               (et affiche)
9      }
10
11     static class monThreadWrite extends Thread {
12         String val;
13         ThreadSafeReadWriteBox b;
14
15         monThreadWrite(ThreadSafeReadWriteBox x, String y) {
16             b = x;
17             val = y;
18         }
19
20         public void run() {
21             b.set(val);
22         }
23     }
24
25     static class monThreadRead extends Thread {
26         ThreadSafeReadWriteBox b;
27
28         monThreadRead(ThreadSafeReadWriteBox x) {
29             b = x;
30         }
31
32         public void run() {
33             try {
34                 System.out.println(b.get());
35             } catch (InterruptedException e) {
36                 System.out.println("Thread interrompu.");
37             }
38         }
39     }
40 }
```

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`.

Correction : Une version avec des fonctions d'ordre supérieur prenant des `Runnable` comme argument, exécuté dans un bloc `try/finally` :

```
1  import java.util.Optional;
2  import java.util.function.Supplier;
3
4  public final class SafeReadWriteLock {
5      private final ReadWriteLock lock = new ReadWriteLock();
6
7      public void read(Runnable r) throws InterruptedException {
8          lock.acquireReaderPrivilege();
9          try {
10             r.run();
11         } finally {
12             lock.dropReaderPrivilege();
13         }
14     }
15
16     public void write(Runnable r) throws InterruptedException {
17         lock.acquireWriterPrivilege();
18         try {
19             r.run();
20         } finally {
21             lock.dropWriterPrivilege();
22         }
23     }
24 }
```

Une autre version sans fonction d'ordre supérieur, mais en retournant des « permis » implémentant `Autocloseable`, ce qui permet l'utilisation dans un bloc *try-with-resource* :

```
1  public final class SafeReadWriteLock2 {
2      private final ReadWriteLock lock = new ReadWriteLock();
3
4      public ReadToken getReadToken() throws InterruptedException {
5          return new ReadToken();
6      }
7
8      public WriteToken getWriteToken() throws InterruptedException {
9          return new WriteToken();
10     }
11
12     public class ReadToken implements AutoCloseable {
13         private ReadToken() throws InterruptedException {
14             lock.acquireReaderPrivilege();
15         }
16
17         @Override
18         public void close() {
```

```
            lock.dropReaderPrivilege();
        }
    }

    public class WriteToken implements AutoCloseable {
        private WriteToken() throws InterruptedException {
            lock.acquireWriterPrivilege();
        }

        @Override
        public void close() {
            lock.dropWriterPrivilege();
        }
    }

}
```

Exemple d'utilisation :

```
public class SRWLDemo {
    public static void main(String[] args) throws InterruptedException {
        SimpleBox y = new SimpleBox();
        SafeReadWriteLock l1 = new SafeReadWriteLock();
        SafeReadWriteLock2 l2 = new SafeReadWriteLock2();


        // utilisation de l1 en lecture :
        l1.read(() -> {
            System.out.println(y.content);
        });
        // utilisation de l1 en écriture :
        l1.write(() -> {
            y.content = "C";
        });

        // utilisation de l2 en lecture (avec try-with-resource) :
        // (équivalent à un try/finally dont le finally appellerait token.close())
        try (SafeReadWriteLock2.ReadToken token = l2.getReadToken()) {
            System.out.println(y.content);
        }

        // utilisation de l2 en écriture (avec try-with-resource) :
        // (équivalent à un try/finally dont le finally appellerait token.close())
        try (SafeReadWriteLock2.WriteToken token = l2.getWriteToken()) {
            y.content = "D";
        }


    }

    static class SimpleBox {
        String content = "Init";
    }
}
```

Autre possibilité : on peut rendre les choses encore plus sûres en forçant les accès en écriture à passer par un permis d'écriture, en faisant en sorte qu'un verrou ne puisse servir qu'à contrôler l'accès à un certain attribut modifiable encapsulé (même principe que `ThreadSafeReadWriteBox`, mais en version générique). Contrairement à `ThreadSafeReadWriteBox` ici, il n'y aurait pas de setteur mais, à sa place, une méthode prenant en paramètre une `Function<T, T>` qui servirait à modifier la donnée encapsulée : `content = f.apply(content)`.

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
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
43     public volatile Data content;
44
45     public Ressource3(int x) { content = new Data(x); }
46 }
```

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.
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 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 » ?

```
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  }
```

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, 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.