Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2019 – 2020
ATL
APA - JLC

ATL – Atelier Logiciel

TD1 – Multithreading

Threads en JAVA

Dans ce document, vous trouverez une introduction au multithreading. Les codes sont disponibles via git à l'adresse :
https://git.esi-bru.be/ATL/Threads.git

Table des matières

Ce document est distribué sous licence
Creative Commons Paternité - Partage à l'Identique 2.0 Belgique (http://creativecommons.org/licenses/by-sa/2.0/be/).
Les autorisations au-delà du champ de cette licence peuvent être demandées à esi-atlg4-list@he2b.be.

Version du document
3 février 2020

1 Préalables

Java est *multi-thread*, c'est-à-dire qu'il est possible d'exécuter, au sein de la même machine virtuelle Java (JVM), donc au sein de la même application, plusieurs fils d'exécution (*thread*) ou séquences d'instructions en parallèle.

La première partie de ce TD relatif à l'étude du multithreading en Java s'attache à introduire quelques notions fondamentales liées au cycle de vie d'une thread [1]. La seconde aborde le problème de la synchronisation et de la communication entre les threads.

Deux types différents de coordination entre fils d'exécutions sont à envisager. Il faut, d'une part, synchroniser l'accès aux données (voir 4 page 11). En effet plusieurs threads peuvent utiliser les mêmes variables et changer leur valeur. Il faut s'assurer que l'accès à ces variables se fassent de manière à ce que ces données restent cohérentes. D'autre part, il faut permettre aux threads de communiquer entre elles, c'est-à-dire de transférer de l'information d'une thread à une autre (voir 3 page 11).

2 Java threading API - Cycle de vie d'une thread

2.1 Création d'une thread

Java intègre dans le langage la possibilité de dédoubler les séquences d'exécutions au sein d'un même programme (processus) dont les différentes ressources leur sont aisément accessibles.

Il existe (au moins) trois manières différentes de créer un fil d'exécution (thread) en JAVA :

1. par dérivation (héritage)
2. par implémentation d'une interface
3. par composition.

Nous n'abordons la création d'une thread par le biais d'une interface que dans ce sous-chapitre. Tous les autres exemples de ce TD font appel à la dérivation ou à la composition.

2.1.1 Création par héritage, la classe `Thread`

Soient la classe *MyThread* qui implémente une thread par héritage et la classe *TestMyThread* qui crée et lance une thread. Consutez et exécutez la classe *TestMyThread*.

On crée une classe Thread qui extends la classe Thread de l'API Java

1. Comment la thread est-elle créée et exécutée ?
2. Quelle est l'utilité de la méthode `start` ? Ou est-elle implémentée ?
3. Changez la valeur maximale de la variable d'itération j de la méthode `run` dans la classe *MyThread*. Que consatez-vous après plusieurs exécutions ?
4. Changez l'appel de la méthode `start` par celui de la méthode `run` dans la classe *TestMyThread*. Quelles sont les conséquences sur l'exécution du programme ?

Montrer que les threads ne sont pas exécutés dans le même ordre en fonction des exécutions. Elle est implémentée dans la classe Thread de l'API java

Plus le nombre est petit, plus les exécutions se ressemblent

Les exécutions sont toutes pareilles

1. Réglons d'emblée le problème du sexe d'un thread ... ou pas, dans la littérature, certains diront **un** thread, d'autre **une** thread. Vous pouvez faire votre choix.

```
$ cat MyThread.java
```

```java
package nvs.alg2ir;

/**
 * Creation d'une classe thread par derivation de la classe Thread
 */
public class MyThread extends Thread {

   private String name;

   public MyThread(String s) {
      this.name = s;
   }

   public void run() {
      for (int i = 0; i < 10; ++i) {
         for (int j = 0; j < 50000; ++j) ;
         System.out.println("MyThread: " + name + " : " + i);
      }
   }
}
```

```
$ cat TestMyThread.java
```

```java
package nvs.alg2ir;

/**
 * Classe de test de la classe MyThread
 */
public class TestMyThread {

   public static void main(String[] args) {
      MyThread t = new MyThread("one");
      t.start();
      // t.run();
      for (int i = 0; i < 10; ++i) {
         for (int j = 0; j < 50000; ++j) ;
         System.out.println("TestMyThread: " + i);
      }
   }
}
```

Création et exécution d'un objet Thread : `start()` et `run()`

- Pour créer une thread, il suffit d'instancier un objet qui étend la classe *Thread* et d'appeler sa méthode `start()`. Cette méthode, héritée de la classe *Thread*, associe les ressources systèmes nécessaires à l'exécution de la thread et démarre son exécution en invoquant sa méthode `run()`. Deux threads sont dés lors exécutées en parallèle : la thread parent (main) et la thread enfant fraichement créée. Il est interdit d'appeler plusieurs fois la méthode `start()` d'un même objet Thread.
- La méthode `run()` contient par réécriture le code à exécuter. Cette méthode correspond à la méthode main de la thread. Elle est supposée se terminer normalement avant la destruction de la thread supportant son exécution.
- Il convient de noter que si la méthode `run()` est directement appelée, celle-ci s'exécute dans la thread courante. Afin de créer un nouveau fil d'exécution, il faut utiliser la méthode `start()`.

Ordre d'exécution des programmes multithreadés

- Contrairement aux programmes rencontrés jusqu'à présent, l'ordre d'exécution des programmes multithreadés n'est pas séquentiel et ne peut dés lors être déterminé à l'avance. En effet, le **scheduler** ou **ordonanceur** du système d'exploitation (et non pas la JVM) gère l'exécution des différentes threads. Il va essayer d'optimiser la parallélisation et donc l'ordre d'exécution des différentes instructions. Une thread peut donc être interrompue pour donner la main à une autre thread à n'importe quel moment de l'exécution de la méthode `run`. Les conséquences que cela implique sont discutées dans une des sections suivantes.
- Ce concept de parallélisme des tâches est déjà bien présent au niveau des systèmes d'exploitation. On parle d'ailleurs de système d'exploitation multitâches gérant l'exécution de plusieurs processus (*process*). Les communications entre les différents processus sont rudimentaires : *pipes*, mémoires partagées.

Java Virtual Machine

2.1.2 Création par implémentation, l'interface `Runnable`

Soient la classe `MyRunnable` qui implémente l'interface `Runnable` et la classe `TestMyRunnable` qui crée et lance une thread. Exécutez la classe `TestMyRunnable`.

1. Comment la thread est-elle créée et exécutée ? Quelles sont les différences avec l'exemple précédent ?

```
$ cat MyRunnable.java
```

```java
package nvs.alg2ir;

/**
 * Creation d'une classe thread par implementation de l'interface Runnable
 */
public class MyRunnable implements Runnable {

   private String name;

   public MyRunnable(String name) {
      this.name = name;
   }

   public void run() {
      for (int i = 0; i < 10; ++i) {
         for (int j = 0; j < 10000000; ++j) ;
         System.out.println("MyRunnable: " + name + " : " + i);
      }
   }
}
```

```
$ cat TestMyRunnable.java
```

```java
package nvs.alg2ir;

/**
 * Classe de test de la classe MyRunnable
 */
public class TestMyRunnable {

   public static void main(String[] args) {
      MyRunnable r = new MyRunnable("one");
      Thread t = new Thread(r);
      t.start();
      for (int i = 0; i < 10; ++i) {
         for (int j = 0; j < 10000000; ++j) ;
         System.out.println("TestMyRunnable: " + i);
      }
   }
}
```

L'interface `Runnable`

▷ L'interface fonctionnelle `Runnable` est une interface possédant une seule méthode : `run()`. Une instance de la classe implémentant cette interface peut être passée comme argument de construction d'un objet `Thread`. La thread exécutera la méthode `run()` après l'appel de sa méthode `start`.

2.1.3 Classe `Thread`, interface `Runnable` et expression lambda

L'interface `Runnable` étant une interface fonctionnelle, il est possible d'instancier une objet `Runnable` via une expression lambda.

```
$ cat TestThreadRunnableLambda.java
```

```java
package nvs.alg2ir;

/**
 * Creation d'une Thread via une expression lambda
 */
public class TestThreadRunnableLambda {

   public static void main(String[] args) {
     // String name = "lambda_";
     new Thread(() -> {
         for (int i = 0; i < 10; ++i) {
            for (int j = 0; j < 50000; ++j) ;
            System.out.println("MyThread: lambda: " + i);
            // System.out.println("MyThread: " + name + ": " + i);
         }
      }).start();

      for (int i = 0; i < 10; ++i) {
         for (int j = 0; j < 50000; ++j) ;
         System.out.println("TestMyThread: " + i);
      }
   }

}
```

2.1.4 Création par composition, un attribut `Thread`

Reprenez l'exemple 2.1.1 et adaptez-le de sorte à définir la classe `MyThreadComposition`, obtenue par composition, en lieu et place de la classe `MyThread`.

2.2 Interruption d'une thread

2.2.1 Méthodes `sleep` et `yield`

La méthode `sleep` est une méthode statique de la classe `Thread` permettant de mettre en sommeil la thread *courante* durant un certain laps de temps.

Soient les classes `MyTimer` et `TestMyTimer`.

1. Remarquez les différents appels à la méthode `sleep()` dans les deux classes. A quel output vous attendez-vous après l'exécution de la méthode main ?
2. Pourquoi l'appel de la méthode `sleep()` différe selon la classe où elle est appelée ?
3. Qu'est ce qu'une `InterruptedException` ?
4. Quand la thread `myTimer` cesse-t-elle de s'exécuter ?

```
$ cat MyTimer.java
```

```java
package nvs.alg2ir;

/**
 * Classe thread affichant un petit message a intervalle regulier : exemple
 * d'utilisation de la methode Thread.sleep
 */
public class MyTimer extends Thread {

   private int laps;
   public boolean shouldRun; // notez le public !

   public MyTimer(int laps) {
      this.laps = laps;
      shouldRun = true;
   }

   public void run() {
      while (shouldRun) {
         try {
            sleep(laps / 2);
            System.out.println("MyTimer: run");
            sleep(laps / 2);
         } catch (InterruptedException e) {
            System.out.println(e);
         }
      }
   }
}
```

```
$ cat TestMyTimer.java
```

```java
package nvs.alg2ir;

/**
 * Classe de test de la classe MyTimer
 */
public class TestMyTimer {

   public static void main(String[] args) {
      MyTimer myTimer = new MyTimer(4000);
      myTimer.start();
      try {
         Thread.sleep(7011);
      } catch (InterruptedException e) {
         System.out.println("TestMyTimer: exception " + e);
      }
      myTimer.shouldRun = false;
      System.gc();
      System.out.println("MyTimer: gc called");
      System.out.println("MyTimer: end");
   }

}
```

Les méthodes `sleep` et `yield`

- La méthode `static sleep(long millis)` de la classe Thread permet de cesser l'exécution de la thread pour une durée de *millis* millisecondes.
- L'exception `InterruptedException` est lancée lorsqu'une thread est en attente, endormie ou occupée et qu'elle est interrompue avant ou pendant son activité.
- Signalons également la méthode statique `yield()` qui met la thread courante en pause, rend la main au *scheduler* (gestionnaire de threads du système d'exploitation) et permet à une autre thread de s'exécuter. Elle est donc équivalente à `sleep(0)`.

2.2.2 Méthode `interrupt()`

Soient les classes `MyTimerInterrupt` et `TestMyTimerInterrupt`.

1. Analysez et exécutez les deux classes. A quoi servent les méthodes `interrupted()` et `interrupt()` ?
2. Exécutez la classe test en ayant auparavant commenté le `return` de la méthode `run`. Que se passe-t-il ? Quelle en est l'explication ?

$ cat MyTimerInterrupt.java

```java
package nvs.alg2ir;

/**
 * Classe thread affichant un petit message a intervalle regulier : exemple
 * d'utilisation de la methode Thread.interrupted
 */
public class MyTimerInterrupt extends Thread {

   private int laps;

   public MyTimerInterrupt(int laps) {
      this.laps = laps;
   }

   @Override
   public void run() {
      while (!interrupted()) {
         try {
            System.out.println("MyTimer: not interrupted");
            sleep(laps);
         } catch (InterruptedException e) {
            System.out.println("MyTimer: exception " + e);
            return;  // essayer avec et sans ce return !
         }
      }
   }
}
```

$ cat TestMyTimerInterrupt.java

```java
package nvs.alg2ir;

/**
 * Classe de test de la classe MyTimerInterrupt : utilisation de la methode
 * interrupt()
 */
public class TestMyTimerInterrupt {

   public static void main(String[] args) {
      MyTimerInterrupt myTimer = new MyTimerInterrupt(1000);
      myTimer.start();
      try {
         Thread.sleep(7011);
      } catch (InterruptedException e) {
         System.out.println("TestMytimer: exception " + e);
      }
      myTimer.interrupt();
   }
}
```

Les méthodes d'interruption

- ▷ La thread dont la méthode `interrupt` est appelée voit son drapeau (indicateur d'état) « interrompu » mis à vrai. Cette méthode n'interrompt ni directement ni brutalement la thread, elle demande à la thread de s'interrompre.
- ▷ La valeur du drapeau « interrompu » peut être consultée au moyen de la méthode d'instance `isInterrupted()`. La valeur du drapeau n'est pas modifiée à cette occasion.
- ▷ Si la thread interrompue était bloquée par `sleep()`, `wait()` ou `join()` [a],
  - ▷ elle est réveillée,
  - ▷ une `InterruptedException` est lancée,
  - ▷ son indicateur d'état « interrompu » **est mis à faux**.
- ▷ D'autre part, la méthode statique `interrupted()` retourne la valeur du drapeau « interrompu » de la thread courante et le remet à faux.

a. Ces deux dernières méthodes concernent la synchronisation des threads. Elles ne sont pas abordées ici.

Remarque

En raison d'une implémentation hasardeuse d'`interrupt()` dans les versions du SDK précédentes à 1.4, l'usage de sémaphores (variables logiques) pour mettre fin en douceur à une thread est privilégié. La même remarque vaut pour les méthodes `stop`, `suspend`, `resume` et `destroy`, désormais obsolètes (*deprecated*).

2.3 Thread utilisateur et démon

Soit la classe `DaemondThread`, analysez et exécutez la.

1. Quel comportement observez-vous ? Il imprime "DaemonThread: run + nombre" 16 fois
2. Que se passe-t-il si vous changez l'appel de la méthode `sleep(7110)` par `sleep(0)` ?
3. Décommentez l'instruction `d.join()`, que se passe-t-il ? Il ne l'imprime qu'une seule fois

Il va attendre que le main crache et le main crache à 42, du coup, il l'imprime 42 fois

```
$ cat DaemonThread.java
```

```java
package nvs.alg2ir;

/**
 * Exemple de thread demon ou utilisateur
 */
public class DaemonThread extends Thread {

   public void run() {
      for (int n = 0; n < 42; ++n) {
         System.out.println("DaemonThread: run " + n);
         try {
            sleep(420);
         } catch (InterruptedException e) {
            System.out.println("DaemonThread thread: exception " + e);
         }
      }
   }

   public static void main(String[] args) {
      DaemonThread d;
      d = new DaemonThread();
      d.setDaemon(true);
      d.start();
      try {
         System.out.println("DaemonThread main: i do nothing during a while");
         sleep(7110);
         // d.join();
      } catch (InterruptedException e) {
         System.out.println("DaemonThread: exception " + e);
      }
   }
}
```

Les threads utilisateur et démon

- Il existe deux catégories de threads : les threads utilisateurs (comme le `main` et toutes les threads rencontrées jusqu'ici) et les démons (*daemons*).
- Lorsque les seules threads en exécution sont des démons, elles sont brutalement arrêtées (attention : à n'importe quel moment de leur `run` !) et l'application prend fin. Les threads utilisateurs perdurent quant à elles jusqu'à la sortie de leur `run`.
- Le type d'une thread est, par défaut, celui de la thread qui l'a créée. Avant l'exécution de la méthode `start` d'une thread, sa méthode `setDaemon(boolean)` permet de fixer sa catégorie.
- Il est possible de demander à une thread d'attendre la fin d'une autre thread. C'est la méthode `join` qui s'en charge.

2.4 Avantages et inconvénients du multithreading

Au même titre que la programmation récursive, le multithreading est un style de programmation qui s'avère très efficace algorithmiquement lorsqu'il est utilisé judicieusement. Il permet, par exemple, d'améliorer les performances d'un programme en limitant les blocages dus aux traitements longs. Les tâches spécifiques peuvent être séparées et s'exécuter chacune dans un fil (affichage de l'interface graphique, impression, téléchargement, etc.).

Il s'agit cependant d'user de cette technique avec précaution. Un nombre trop important de *threads* risque en effet de dégrader les performances en demandant beaucoup de traitements au processeur. Les données étant partagées par toutes les threads d'un *process*, il faut veiller à préserver leur cohérence en « synchronisant » leurs accès par les fils d'exécutions concurrents. Le débogage ne sort pas indemne de la programmation multi-fils, l'ordre d'exécution des différentes threads n'étant pas contrôlé absolument.

Remarque

Le multithreading n'est pas géré directement par la JVM mais par l'interface native des threads du système d'exploitation. Voilà pourquoi il n'y a pas de JVM pour les systèmes d'exploitation qui ne supportent pas le multithreading (Ms-Dos, par exemple). Ceci implique également qu'une application à plusieurs fils d'exécution peut se comporter différemment selon le système d'exploitation sous-jacent. Ces différences sont assez mineures et ne devraient affecter que les performances.

3 Coordination entre threads

Il peut advenir qu'une thread ne puisse poursuivre son exécution avant qu'une seconde thread ait réalisé une tâche spécifique. Les threads, bien que concurrentes, peuvent être amenées à devoir collaborer, elles sont amenées à attendre des signaux provenant d'autres threads [2].

Les méthodes `wait()`, `notify()` et `join()` servent à faire face à ce type de situation (voir l'exemple de la section 2.3).

Il peut également advenir que des threads doivent communiquer et donc partager un certain canal de communication. Java permet la communication entre différents threads *via* les *pipes* [3] (voir `java.nio.channels.Pipe`).

Ces concepts ne sont pas abordés dans ce TD [4].

4 Synchronisation entre threads

4.1 Illustration du problème d'accès concurrent

Soient les trois classes `ToujoursPair`, `MyThread` et `Test`. Analysez ces trois classes et exécutez la classe `Test`.

1. Que constatez-vous ? Quelle est la cause de ce comportement ?

2. Dans le cas de processus, les processus s'envoient des signaux, voir par exemple, la fonction C, `signal`
3. Dans le cas de processus, les processus communiquent *via* un `pipe`
4. Et c'est dommage ;-)

$ cat ToujoursPair.java

```java
package nvs.alg2ir.concurrent;

/**
 * Petite classe pourvue de deux methodes simples
 *
 * Exemple inspire par Thinking in Java, 3rd Edition, Beta Copyright (c)2002 by
 * Bruce Eckel www.BruceEckel.com
 */
public class ToujoursPair {

   private int i = 0;

   public void nextI() {
      ++i;
      ++i;
   }

   public int getI() {
      return i;
   }
}
```

$ cat MyThread.java

```java
package nvs.alg2ir.concurrent;

/**
 * Thread accedant en lecture a une instance de ToujoursPair
 *
 * Exemple inspire par Thinking in Java, 3rd Edition, Beta Copyright (c)2002 by
 * Bruce Eckel www.BruceEckel.com
 */
public class MyThread extends Thread {

   ToujoursPair tp;

   public MyThread(ToujoursPair tp) {
      this.tp = tp;
   }

   public void run() {
      while (true) {
         int val = tp.getI();
         if (val % 2 != 0) {
            System.out.println("myThread : " + val);
            System.exit(0);
         }
      }
   }
}
```

```
$ cat Test.java
```

```java
package nvs.alg2ir.concurrent;

/**
 * Thread accedant en ecriture et lecture a une instance de ToujoursPair
 *
 * Exemple inspire par Thinking in Java, 3rd Edition, Beta Copyright (c)2002 by
 * Bruce Eckel www.BruceEckel.com
 */
public class Test {

   public static void main(String[] args) {
      ToujoursPair tp = new ToujoursPair();
      MyThread t = new MyThread(tp);
      t.start();
      while (true) {
         tp.nextI();
         if (tp.getI() % 1000000 == 0) {
            System.out.println(tp.getI());
         }
      }
   }
}
```

Les threads utilisateur et démon

- La grande différence entre processus et threads est que ces dernières s'exécutent au sein d'un même programme. Cela implique que certains objets, certaines données sont automatiquement partagées par les différentes threads du programme. Dans l'exemple ci-dessus, la variable `tp`.
- Ce partage doit être contrôlé de sorte que deux threads accédant au même objet le font de manière cohérente ou, en d'autres termes, qu'une thread ne laisse ni ne trouve jamais un objet dans un état incohérent. Ainsi, si l'une modifie un attribut (`nextI()`) tandis que l'autre le consulte (`getI()`), il faut veiller à ce que les deux accès ne se chevauchent pas. Rappelons en effet que la méthode `run()` peut être interrompue *à tout moment* par le scheduler.

4.2 Méthode `synchronized`

Pour pallier aux désagréments engendrés par le comportement mis en évidence dans l'exemple précédent, on peut utiliser le mot clé `synchronized` comme **modificateur de méthode**.

Les classes `MyObjet`, `MyThread` et `Test` illustrent une utilisation de méthodes `synchronized`.

1. Après exécution, observez les entrées et sorties des méthodes `synchronized`, qu'observez-vous ?
2. Testez les quatre combinaisons des signatures des méthodes `show` et `print`.
3. Décommentez les commentaires relatifs à `mo2` et `mt2`.

Un PROCESSUS ne peut pas être interrompu pendant son exécution. Donc, un processus2 doit attendre la fin d'un processus1 pour pouvoir commencer à être exécuté.
Un THREAD est un fil d'exécution qui peut être interrompu pendant son exécution afin de permettre à 2 fils d'être exécutés "en même temps". A vrai dire, ils s'exécutent un peu l'un, puis un peu l'autre, puis un peu l'un,... selon ce que le processeur choisit.

```
$ cat MyObject.java
```

```java
package nvs.alg2ir.synchronised;

/**
 * Classe pourvue de deux methodes d'affichage : illustration de l'utilisation
 * du mot cle synchronized comme modificateur de methode.
 */
public class MyObject {

   private String name;

   public MyObject(String name) {
      this.name = name;
   }

   // public void show() {
   public synchronized void show() {
      String nom = Thread.currentThread().getName();
      System.out.println("My object: thread " + nom
            + ", objet  " + name + " in show");
      try {
         Thread.sleep(7000);
      } catch (InterruptedException e) {
      }
      System.out.println("My object: thread  " + nom
            + ",objet " + name + " out show");
   }

   // public void print() {
   public synchronized void print() {
      String nom = Thread.currentThread().getName();
      System.out.println("My object: thread  " + nom
            + ", objet " + name + " in print");
      try {
         Thread.sleep(7000);
      } catch (InterruptedException e) {
      }
      System.out.println("My object: thread " + nom
            + ", objet " + name + " out print");
   }
}
```

$ cat MyThread.java

```java
package nvs.alg2ir.synchronised;

/**
 * Thread utilisant une methode d'une instance de MyObject
 */
public class MyThread extends Thread {

   private MyObject myObject;

   public MyThread(String name, MyObject myObject) {
      super(name);
      this.myObject = myObject;
   }

   public void run() {
      String nom = Thread.currentThread().getName();
      System.out.println("My thread: thread  " + nom + " in run");
      myObject.show();
      System.out.println("My thread: thread " + nom + " out run");
   }
}
```

$ cat Test.java

```java
package nvs.alg2ir.synchronised;

/**
 * Thread utilisant une methode d'une instance de MyObject et instanciant une ou
 * deux nouvelles threads.
 */
public class Test {

   public static void main(String[] args) {
      MyObject mo1 = new MyObject("mo1");
      // MyObject mo2 = new MyObject("mo2");
      MyThread mt1 = new MyThread("mt1", mo1);
      // MyThread mt2 = new MyThread("mt2", mo2);

      mt1.start();
      // mt2.start();
      try {
         Thread.sleep(0L);
      } catch (InterruptedException e) {
      }
      mo1.print();
   }
}
```

Les threads utilisateur et démon

- L'effet du mot-clé `synchronized` peut être décrits de la manière suivante. Chaque objet en Java possède un verrou et une seule clé ouvrant ce verrou.
  - Pour exécuter une *méthode d'instance non* ***synchronized*** d'un objet donné, une thread ne doit **pas** posséder la clé de cet objet.
  - Par contre, si cette *méthode* est ***synchronized***, la thread qui tente de l'exécuter ne le peut que si la *clé* de l'objet est *disponible*. Si ce n'est pas le cas, elle attend que la clé soit accessible. Si la clé est disponible, elle s'en empare, lance l'exécution de la méthode, puis rend la clé lorsqu'elle retourne de cette méthode.
  - Ceci implique que deux méthodes d'instance `synchronized` d'un même objet, ne peuvent pas être exécutées simultanément par deux threads. Les méthodes peuvent être différentes pour chaque thread, mais il peut également s'agir de la même méthode pour les deux.
- Java propose également la classe `Sémaphore` (voir `java.util.concurrent.Semaphore`) pour la gestion de ressources partagées.

  L'usage du mot clé `volatile` et de la classe `Sémaphore` ne sont pas abordés ici.

Lorsqu'une méthode d'instance est synchronized, elle donnera la main à une autre méthode via la clé de l'objet qu'elle manipule. Donc, le thread doit posséder cette clef.

Remarque

Lors d'appels imbriqués de méthodes synchronisées, la thread ne rend la clé du verrou que lorsqu'elle sort de la méthode la plus *externe*. On parle de *verrouillage réentrant*.

4.3 Bloc `synchronized`

Un Mutex M.M.S est une primitive de synchronisation utilisée en programmation informatique pour éviter que des ressources partagées d'un système ne soient utilisées en même temps.

La portée du mot clé `synchronized` peut être restreinte à un bloc. Dans ce cas il faut fournir en argument l'objet dont la clé du verrou est nécessaire pour y pénétrer. C'est objet peut être comparé au *mutex*, le sémaphore d'exclusion mutuel.

Testez l'exemple suivant en modifiant les objets de synchronisation des blocs `synchronized` de `fct` (quatre combinaisons différentes sont possibles).

La méthode yield() applique une pause à l'exécution du thread courant, afin de libérer le processeur pour d'autres unités d'exécution en attente. A l'instar de la méthode sleep(), la méthode yield() étant déclarée statique, ne peut être invoquée sur une instance de la classe Thread.

```
$ cat MyObject.java
```

```java
package nvs.alg2ir.block;

import java.util.Random;

/**
 * Classe pourvue d'une methode d'affichage avec des blocs synchronized sur
 * l'objet lui-meme ou sur une chaine de caracteres.
 */
public class MyObject {

   private Random rnd;

   public MyObject() {
      rnd = new Random();
   }

   public void fct() {
      String nom = Thread.currentThread().getName();
      System.out.println("MyObject: " + nom + " in fct");

      synchronized (this) {
         //synchronized("verrou") {
         System.out.println("MyObject: " + nom + " in bloc 1");
         try {
            Thread.sleep(rnd.nextInt(1000));
         } catch (InterruptedException e) {
         }
         System.out.println("MyObject: " + nom + " out bloc 1");
      }

      System.out.println("MyObject: " + nom + " between bloc 1 and bloc 2");
      try {
         Thread.sleep(rnd.nextInt(1000));
      } catch (InterruptedException e) {
      }

      synchronized (this) {
      // synchronized ("verrou") {
         System.out.println("MyObject: " + nom + " in bloc 2");
         try {
            Thread.sleep(rnd.nextInt(1000));
         } catch (InterruptedException e) {
         }
         System.out.println("MyObject: " + nom + " out bloc 2");
      }

      System.out.println("MyObject: " + nom + " out fct");
   }
}
```

```
$ cat MyThread.java
```

```java
package nvs.alg2ir.block;

/**
 * Thread utilisant une methode d'une instace de MyObject.
 */
public class MyThread extends Thread {

   private MyObject myObject;

   public MyThread(String name, MyObject myObject) {
      super(name);
      this.myObject = myObject;
   }

   public void run() {
      String nom = Thread.currentThread().getName();
      System.out.println("MyThread: " + nom + " in run");
      myObject.fct();
      System.out.println("MyThread: " + nom + " out run");
   }
}
```

```
$ cat Test.java
```

```java
package nvs.alg2ir.block;

/**
 * Classe de test instanciant deux threads utilisant une methode a blocs
 * synchronises d'un meme objet.
 */
public class Test {

   public static void main(String[] args) {

      MyObject mo = new MyObject();
      MyThread t1 = new MyThread("t1", mo);
      MyThread t2 = new MyThread("t2", mo);

      t1.start();
      Thread.yield();
      t2.start();
   }
}
```

Remarque

Lorsque la synchronisation se fait sur une chaîne de caractères, on parle de *synchronisation nommée*.
La synchronisation peut également se faire sur n'importe quel objet.

4.4 Étreinte mortelle (*deadlock*)

L'étreinte mortelle[5] (*deadlock*) survient lorsque deux threads sont en attente d'une ressource que possède l'autre thread. C'est-à-dire ;

- ▷ la thread 1 possède la clé de l'objet A et attend celle de l'objet B,

5. Cette expression est digne d'un mauvais film d'horreur, on lui préfère le vocable *interbloquage*.

- ▷ la thread 2 possède la clé de l'objet B et attend celle de l'objet A

Ce type d'erreur est difficile à détecter.

Exercice 1 Écrivez un code générant un *deadlock*.

Exercice 2 Écrivez une classe `Compte` qui permet de maintenir un solde bancaire en ayant 3 méthodes :

- ▷ `deposerArgent(int somme)`,
- ▷ `retirerArgent(int somme)`
- ▷ et `int getSomme()`.

Écrivez ensuite une classe `Operation` qui simule, dans une thread, une succession infinie d'opérations ajoutant et retirant une même somme sur un compte donné et puis affichant le solde du compte.

Écrivez ensuite une classe `Main` qui fait tourner 20 threads de type `Operation` en même temps sur un même objet du type `Compte`.

Corrigez si nécessaire votre code jusqu'à ce que le solde du compte soit cohérent après chaque opération (jamais de solde négatif par exemple).

5 Threads et JavaFx

Lors de l'exécution d'une application une thread principale est créée. Toutefois via l'utilisation des classes du package `java.util.concurrent` il est possible de créer d'autres threads pour permettre la parallélisation de fils d'exécutions.

Une application *JavaFX* hérite de la classe `Application`. Pour lancer une telle application il faut exécuter la méthode statique `Application.launch()`.

Cette méthode :

- ▷ appelle la méthode d'initialisation `Application.init()` qui est exécutée dans la thread principale ;
- ▷ appelle la méthode `Application.start()` qui crée la `JavaFX Application Thread` dont l'objectif est de gérer l'ensemble des actions de l'interface graphique ;

Au sein de cette interface un utilisateur peut exécuter une tâche d'une longue durée : chargement d'images, requêtes sur une base de données, calcul complexe, sauvegarde d'un fichier, ...

Si ces opérations sont traitées au sein du `JavaFX Application Thread` l'interface sera gelée jusqu'à la fin de cette tâche. Dés lors ces opérations doivent s'effectuer en tâches de fonds et, si nécessaire, afficher une barre de progression pour informer l'utilisateur de l'état d'avancement de la tâche.

Cette gestion peut s'effectuer via les classes du package `java.util.concurrent`, mais pour faciliter le travail du développeur il existe des classes dédiées à ce travail au sein du package `javafx.concurrent`, dont voici le tutoriel [6]

6. https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm