

# TD4<sup>1</sup>

## Les threads

### 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*) 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<sup>2</sup>. La seconde aborde le problème de la synchronisation et de la communication entre les threads.

Deux types différents de coordinations entre fils d'exécutions sont à envisager. Il faut, d'une part, synchroniser l'accès aux données (voir 5 page 10) et, d'autre part, permettre aux threads de communiquer entre elles, c'est-à-dire de transférer de l'information d'une thread à une autre (voir 4 page 9).

### 2 Le concept de thread (fil d'exécution)

Le texte de cette section est largement inspiré par l'introduction aux threads de *Java : La maîtrise*, Jérôme Bougeault, Tsoft/Eyrolles, 2003.

#### 2.1 Thread et processus

Classiquement le déroulement d'un programme (sans interface graphique) se fait séquentiellement à partir de son point d'entrée. Cependant, certaines parties du programme peuvent parfois être exécutées parallèlement. Une séquence d'exécution peut alors être initialisée pour chacune de ces parties.

---

<sup>1</sup>Le texte de ce TD s'inspire très largement de ceux des ateliers logiciels écrits jadis par différents collègues. Parmi les TDs que comprend ce cours, *MCD*, *NVS*, *RFS*, *SMB* et *VAK* ont contribué de manière diverses. Certains ont écrits entièrement un TD, d'autres ont contribué ou coécrits un TD, d'autres encore ont fait du travail de relecture. Je (*PBT*) laisse mon empreinte et en profite pour remercier tous ceux qui aiment être remerciés.

<sup>2</sup>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.

Ce concept de parallélisme des tâches est déjà bien présent au niveau des systèmes d'exploitation. On parle d'ailleurs de système d'exploitation multi-tâches gérant l'exécution de plusieurs processus (*process*). Les communications entre les différents processus sont rudimentaires : *pipes*, mémoires partagées.

Java intègre (dans le langage) la possibilité de dédoubler les séquences d'exécutions au sein d'un même programme, c'est-à-dire au sein d'un même processus. Comme les multiples threads d'une application s'exécutent dans le même *process*, les ressources de ce processus leurs sont aisément accessibles. On a ainsi un partage global des objets issus des différentes threads s'exécutant sur une machine virtuelle.

## 2.2 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-fil, l'ordre d'exécution des différentes thread 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 Java threading API - Cycle de vie d'une thread

### 3.1 Création d'une thread

Il existe (au moins) deux manières différentes de créer un fil d'exécution. Soit par dérivation (héritage), soit par implémentation d'une interface. Remarquez cependant qu'une troisième solution est fournie par la composition de classes. Nous n'aborderons 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.

#### 3.1.1 Création par héritage, la classe Thread

La classe Thread définit, entre autre, les méthodes `run()` et `start()`. La méthode `run` contient, par réécriture, le code à exécuter. Cette méthode correspond à la méthode `main` d'une application. Elle est supposée se terminer normalement avant la destruction de la thread supportant son exécution. La méthode `start` permet de démarrer l'exécution de la thread, ce que celle-ci fait en invoquant `run`.

Comme déjà signalé, ce n'est pas la JVM qui gère l'exécution des threads. Un des points essentiels à retenir de ce TD est que la mise en pause du `run` d'une thread pour donner la main à une autre est réalisée par le gestionnaire de thread du système d'exploitation (*scheduler*) et peut avoir lieu à *n'importe quel moment de l'exécution du `run`* de ce fil d'exécution !

**Exemple** Testez l'exemple suivant : remplacez l'appel de `start` par `run`, modifiez les valeurs maximales des compteurs de boucle.

```
$ cat MyThread
```

```
package nvs.alg2ir;

/**
 * Création d'une classe thread par dérivation 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 < 10000000 ; ++j) ;
            System.out.println("MyThread:_ " + name + " :_ " + i) ;
        }
    }
}
```

```
$ cat TestMyThread
```

```
package nvs.alg2ir;

/**
 * Classe de test de la classe UneThread
 */
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 < 5000000 ; ++j) ;
                System.out.println("TestMyThread:_" + i) ;
            }
        }
    }
}
```

### 3.1.2 Création par implémentation, l'interface Runnable

Cette interface ne possède qu'une seule méthode : `run()`. Cette méthode contient le code à exécuter en parallèle. Une instance de la classe implémentant `Runnable` est passée comme argument de construction d'une thread qui exécute le code de la méthode `run` après appel de `start`.

```
MyRunnable r = new MyRunnable() ;
Thread t = new Thread(r) ;
t.start() ;
```

**Exemple** Testez l'exemple suivant : modifiez les valeurs maximales des compteurs de boucle.

```
$ cat MyRunnable
```

```
package nvs.alg2ir;

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

    private String name ;

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

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

```
$ cat TestMyRunnable
```

```
package nvs.alg2ir;

/**
 * Classe de test de la classe UneRunnable
 */
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) ;
        }
    }
}
```

### 3.1.3 Création par composition, un attribut Thread

Reprenez l'exemple 3.1.1 et adaptez-le de sorte à définir la classe `MyThreadComposition`, obtenue par composition, en lieu et place de la classe `MyThread`.

## 3.2 Interruption d'une thread

### 3.2.1 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 laps de temps.

**Exemple** Testez l'exemple suivant. Quand la thread `myTimer` cesse-t-elle d'exister ?

```
$ cat MyTimer
```

```
package nvs.alg2ir;

/**
 * Classe thread affichant un petit message à intervalle régulier :
 * exemple d'utilisation de la méthode 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

```
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");
    }
}
```

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)`.

### 3.2.2 Méthode `interrupt()`

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()`<sup>3</sup>,  
↪ elle est réveillée,  
↪ une `InterruptedException` est lancée et  
↪ 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.

**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*).

**Exemple** Testez l'exemple suivant avec et sans le `return` dans la méthode `run`.

```
$ cat MyTimer
```

```
package nvs.alg2ir;

/**
 * Classe thread affichant un petit message à intervalle régulier :
 * exemple d'utilisation de la méthode Thread.interrupted
 */
public class MyTimer extends Thread{

    private int laps;

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

    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 !
            }
        }
    }
}
```

---

<sup>3</sup> Ces deux dernières méthodes concernent la synchronisation des threads. Elles ne sont pas abordées ici.

```
$ cat TestMyTimer
```

```
package nvs.alg2ir;

/**
 * Classe de test de la classe MyTimer :
 * utilisation de la méthode interrupt()
 */
public class TestMyTimer{

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

### 3.3 Thread 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.

**Exemple** Testez la classe `DaemonThread` en supprimant le `sleep` de la méthode `main` par exemple. Décommentez et commentez l'instruction `join`.

```
$ cat DaemonThread
```

```
package nvs.alg2ir;

/**
 * Exemple de thread démon 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(70110) ;
            //d.join() ;
        } catch (InterruptedException e) {
            System.out.println("DaemonThread:_exception_" + e);
        }
    }
}
```

## 4 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<sup>4</sup>.

Les méthodes `wait()`, `notify()` et `join()` servent à faire face à ce type de situation (voir l'exemple de la section 3.3).

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*<sup>5</sup> (voir `java.nio.channels.Pipe`).

Ces concepts ne sont pas abordés dans ce TD<sup>6</sup>.

---

<sup>4</sup>Dans le cas de processus, les processus s'envoient des signaux, voir par exemple, la fonction C, `signal`

<sup>5</sup>Dans le cas de processus, les processus communiquent *via* un pipe

<sup>6</sup>Et c'est dommage;-)

## 5 Synchronisation entre threads

Comme mentionné dans la section 2.1, 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.

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 tandis que l'autre le consulte, il faut veiller que les deux accès ne se chevauchent pas. Rappelons en effet que la méthode `run()` peut être interrompue à *tout moment* par le scheduler.

Java propose le mot clé `synchronized` afin de répondre à ce problème de synchronisation de l'accès aux données. 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.

### 5.1 Illustration du problème d'accès concurrent

**Exemple** Testez cet exemple illustrant le problème d'accès concurrent. Que constatez-vous ? Quelle en est la cause ?

```
$ cat ToujoursPair
```

```
package nvs.alg2ir;

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

    private int i = 0 ;

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

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

```
$ cat MyThread
```

```
package nvs.alg2ir;

/**
 * Thread accédant en lecture à une instance de ToujoursPair
 *
 * Exemple inspiré 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
```

```
package nvs.alg2ir;

/**
 * Thread accédant en écriture et lecture à une instance
 * de ToujoursPair
 *
 * Exemple inspiré 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()) ;
            }
        }
    }
}
```

## 5.2 Méthode synchronized

Pour pallier aux désagréments engendrés par le comportement mis au jour dans l'exemple précédent, on peut utiliser le mot clé `synchronized` comme

### modificateur de méthode.

La signification et l'effet de ce mot clé peuvent 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.

Pour vous en convaincre, reprenez l'exemple précédent en affublant certaines (lesquelles?) méthodes de ToujoursPair du mot *synchronized*. Qu'observez-vous? Qu'en est-il de la performance?

Voyons plus en détail l'effet du mot clé *synchronized*.

**Exemple** Testez l'exemple suivant en essayant les quatre combinaisons de signatures des méthodes `show` et `print`, puis en retirant les commentaires relatifs à `mo2` et `mt2`.

```
$ cat MyObject
```

```
package nvs.alg2ir;

/**
 * Classe pourvue de deux méthodes d'affichage :
 * illustration de l'utilisation du mot clé synchronized
 * comme modificateur de méthode.
 */
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

```

package nvs.alg2ir;

/**
 * Thread utilisant une méthode 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

```

package nvs.alg2ir;

/**
 * Thread utilisant une méthode d'une instance de MyObject
 * et instanciant une ou deux nouvelles threads.
 */
public class Test{
    public static void main(String[] args){
        MyObject mol = new MyObject("mol") ;
        //MyObject mo2 = new MyObject("mo2") ;
        MyThread mt1 = new MyThread("mt1",mol) ;
        //MyThread mt2 = new MyThread("mt2",mo2) ;

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

```

**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*.

### 5.3 Bloc synchronized

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.

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

\$ cat MyObject

```
package nvs.alg2ir;

import java.util.Random;

/**
 * Classe pourvue d'une méthode d'affichage avec des blocs
 * synchronized sur l'objet lui-même ou sur une chaîne de
 * caractères.
 */
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

```

package nvs.alg2ir;

/**
 * Thread utilisant une méthode 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("MyThread:_" + nom + "_in_run") ;
        myObject.fct() ;
        System.out.println("MyThread:_" + nom + "_out_run") ;
    }
}

```

\$ cat Test

```

package nvs.alg2ir;

/**
 * Classe de test instanciant deux threads utilisant une méthode à
 * blocs synchronisés d'un même 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() ;
    }
}

```

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, on pourra écrire quelque chose du style :

```
public static Object myObjectLock = new Object() ;
...
synchronized (myObjectLock) {...}
```

## 5.4 L'étreinte mortelle (*deadlock*)

L'étreinte mortelle<sup>7</sup> (*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,

↪ 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** Créez un composant `JBlinkLabel`. Il s'agit d'un `JLabel` dont le texte clignote tandis que ses couleurs d'avant et d'arrière plan changent aléatoirement au même rythme. Le texte à afficher, les couleurs d'avant plan, les couleurs d'arrière plan et le délai de clignotement sont fournis par le constructeur. Si le délai est nul, le texte ne clignote pas ; si un tableau de `Color` est `null`, la couleur par défaut (du plan correspondant) est prise.

Qu'observez-vous avec la `preferredSize` du composant à haute fréquence de clignotement ? Comment résoudre ce problème ?

**Exercice 3** Reprenez l'exercice du *poulailler*<sup>8</sup>

Vous allez maintenant vous arranger pour que chaque poule du poulailler soit une *thread*. Chacune de ces *threads* a un *run* qui gère sa vie. Toutes ces poules partagent un même enclos et un même tas de graines ... elles partagent aussi un même GUI.

La classe `TasGraines` ne présente aucune difficulté. La classe `Enclos`, avec ses deux attributs `JPanel [] []` et `Poule [] []`, permet de spécifier que l'enclos sera partagé entre les poules et le GUI.

La classe `Config` permet de rassembler dans un même fichier tous les paramètres de configuration du projet. C'est plus facile pour les ajuster.

La classe `Poule` hérite de `Thread`. Chaque poule sera un thread. Les poules partagent l'enclos et le tas de graines. Si une poule ne mange pas (insuffisamment de graines dans le tas), elle meurt. Si elle mange, sa vie se résume à ; manger, bouger, éventuellement pondre (si elle a l'âge), attendre un peu et passer au tour suivant. Lorsqu'elle pond, une poule crée quelques poules, elle lance les threads

<sup>7</sup>Cette expression est digne d'un mauvais film d'horreur, on lui préfère le vocable *interblocage*.

<sup>8</sup>Encore lui, nous le traînerons tout au long des Tds.

associés aux poules créées. Certaines des méthodes (ou blocs) de cette classe devront être **synchronisés**.

La classe `Main` s'occupe de l'instanciation du GUI ainsi que des variables partagées, un `TasGraines` et un `Enclos`. Ces deux variables peuvent être utilisées pour la synchronisation des données. Un moniteur (*monitor*) sera créé sur base de l'objet et servira à la synchronisation. Si un moniteur est créé sur l'objet `enclos` par exemple, les méthodes (`synchronized`) **et** les blocs (`synchronized`) sur cet objet seront synchronisés.

Une page de discussion pour cet exercice se trouve sur le wiki -> <http://wiki.namok.be/?id=esi:alg2ir-poulailler>

# Table des matières

<b>1</b>	<b>Préalables</b>	<b>1</b>
<b>2</b>	<b>Le concept de thread (fil d'exécution)</b>	<b>1</b>
2.1	Thread et processus . . . . .	1
2.2	Avantages et inconvénients du multithreading . . . . .	2
<b>3</b>	<b>Java threading API - Cycle de vie d'une thread</b>	<b>3</b>
3.1	Création d'une thread . . . . .	3
3.1.1	Création par héritage, la classe <code>Thread</code> . . . . .	3
3.1.2	Création par implémentation, l'interface <code>Runnable</code> . . . . .	4
3.1.3	Création par composition, un attribut <code>Thread</code> . . . . .	5
3.2	Interruption d'une thread . . . . .	5
3.2.1	Méthodes <code>sleep</code> et <code>yield</code> . . . . .	5
3.2.2	Méthode <code>interrupt()</code> . . . . .	6
3.3	Thread utilisateur et démon . . . . .	8
<b>4</b>	<b>Coordination entre threads</b>	<b>9</b>
<b>5</b>	<b>Synchronisation entre threads</b>	<b>10</b>
5.1	Illustration du problème d'accès concurrent . . . . .	10
5.2	Méthode <code>synchronized</code> . . . . .	11
5.3	Bloc <code>synchronized</code> . . . . .	14
5.4	L'étreinte mortelle ( <i>deadlock</i> ) . . . . .	16

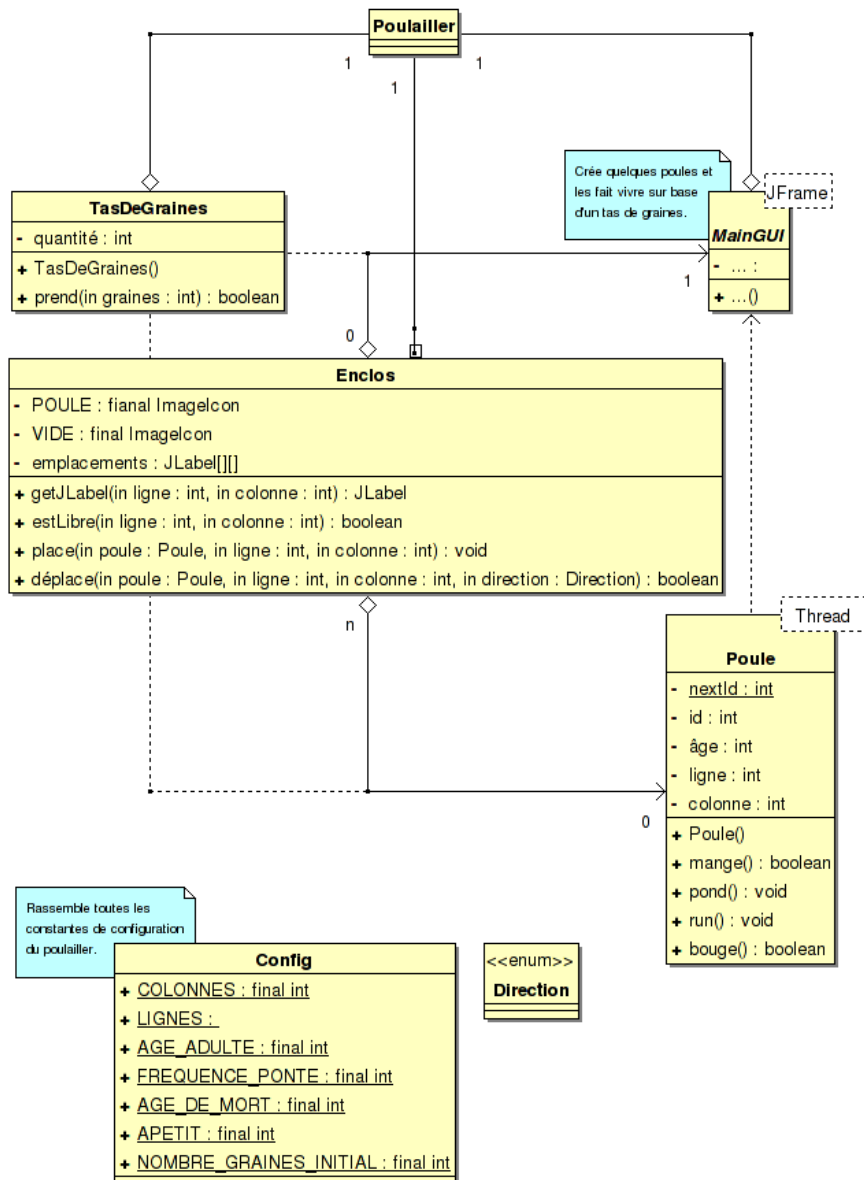


FIG. 1 – Diagramme UML