



ALG2ir-TD04 : Thread

1 Présentation du TD

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. La seconde aborde le problème de la synchronisation.

Deux types différents de coordinations entre fils d'exécutions sont à envisager. Il faut, d'une part, synchroniser l'accès aux données et, d'autre part, synchroniser les threads entre elles. Seule la première synchronisation est étudiée ici.

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 process

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.

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.

¹ Ce TD compte également MCD et RFS parmi ses auteurs.

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.). D'autre part, cette technique de programmation trouvera très probablement dans l'avenir une grande efficacité sur les machines multi-processeurs.

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.

Une remarque suite à ce dernier point. 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

3.1.1 La classe Thread

Il existe (au moins) deux manières différentes de créer un fil d'exécution. Soit par dérivation, soit par implémentation d'une interface. Cette seconde technique est offerte pour contourner l'impossibilité de l'héritage multiple en Java. 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.

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

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Création d'une classe thread par dérivation de la classe Thread
5  */
6 public class UneThread extends Thread {
7
8     private String nom ;
9
10    public UneThread(String nom) {
11        this.nom = nom ;
12    }
13
14    public void run() {
15        for (int i = 0 ; i < 10 ; ++i) {
16            for (int j = 0 ; j < 10000000 ; ++j) ;
17            System.out.println(nom+" : "+i) ;
18        }
19    }
20 }

```

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Classe de test de la classe UneThread
5  */
6 public class Test {
7
8     public static void main(String [] args) {
9         UneThread t = new UneThread("t") ;
10        t.start() ;
11        for (int i = 0 ; i < 10 ; ++i) {
12            for (int j = 0 ; j < 10000000 ; ++j) ;
13            System.out.println("main : "+i) ;
14        }
15    }
16 }

```

3.1.2 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`.

Exemple 2 : Testez l'exemple suivant : modifiez les valeurs maximales des compteurs de boucle.

```

1 package nvs.alg2ir.td04thread.enonce;

```

```

2
3 /**
4  * Création d'une classe thread par implémentation de l'interface
5  * Runnable
6  */
7 public class UneRunnable implements Runnable {
8
9     private String nom ;
10
11    public UneRunnable(String nom) {
12        this.nom = nom ;
13    }
14    public void run() {
15        for (int i = 0 ; i < 10 ; ++i) {
16            for (int j = 0 ; j < 10000000 ; ++j) ;
17            System.out.println(nom+" : "+i) ;
18        }
19    }
20 }

```

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Classe de test de la classe UneRunnable
5  */
6 public class Test {
7
8     public static void main(String [] args) {
9         UneRunnable r = new UneRunnable("r") ;
10        Thread t = new Thread(r) ;
11        t.start() ;
12        for (int i = 0 ; i < 10 ; ++i) {
13            for (int j = 0 ; j < 10000000 ; ++j) ;
14            System.out.println("main : "+i) ;
15        }
16    }
17 }

```

Ex1. Reprenez l'Exemple 1 et adaptez-le de sorte à définir la classe `UneThreadComposee`, obtenue par composition, en lieu et place de la classe `UneThread`.

3.2 Mort d'une thread et mise en sommeil

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

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Classe thread affichant un petit message à intervalle régulier :
5  * exemple d'utilisation de la méthode Thread.sleep

```

```

6  */
7  public class MyTimer extends Thread {
8
9      private Test test;
10     private int laps;
11     public boolean shouldRun; // notez le public !
12
13     public MyTimer(Test test , int laps) {
14         this.test= test;
15         this.laps= laps;
16         shouldRun= true;
17     }
18
19     public void run() {
20         while(shouldRun) {
21             try {
22                 sleep(laps/2);
23                 test.print();
24                 sleep(laps/2);
25             } catch(InterruptedException e) {
26                 System.out.println(e);
27             }
28         }
29     }
30 }

```

```

1  package nvs.alg2ir.td04thread.enonce;
2
3  /**
4   * Classe de test de la classe MyTimer
5   */
6  public class Test {
7
8      public static void main(String[] args) {
9          Test test= new Test() ;
10         MyTimer myTimer= new MyTimer(test,900) ;
11         myTimer.start() ;
12         try {
13             Thread.sleep(7011);
14         } catch(InterruptedException e) {
15             System.out.println(e);
16         }
17         //myTimer.shouldRun = false;
18         myTimer = null;
19         /*
20         try {
21             Thread.sleep(100);
22         } catch(InterruptedException e) {
23             System.out.println(e);
24         }
25         */
26         System.gc() ;
27         System.out.println("gc appelé");
28         /*
29         try {

```

```

30     Thread.sleep(100);
31     } catch (InterruptedException e) {
32         System.out.println(e);
33     }
34     */
35     System.out.println("fin");
36 }
37
38 public void print() {
39     System.out.println("coucou");
40 }
41 }

```

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(0L)`.

3.2.1 Variante avec la 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.

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()`², 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.

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

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Classe thread affichant un petit message à intervalle régulier :
5  * exemple d'utilisation de la méthode Thread.interrupted
6  */
7 public class MyTimer extends Thread{
8
9     private Test test;
10    private int laps;
11    public MyTimer( Test test , int laps){
12        this.test= test;
13        this.laps= laps;

```

² Ces deux dernières méthodes concernent la synchronisation des threads. Elles ne sont pas abordées ici.

```

14 }
15
16 public void run(){
17     while(!interrupted()){
18         try{
19             test.print();
20             sleep(laps);
21         }catch(InterruptedException e){
22             System.out.println(e);
23             return; // essayer avec et sans ce return !
24         }
25     }
26 }
27 }

```

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Classe de test de la classe MyTimer :
5  * utilisation de la méthode interrupt()
6  */
7 public class Test{
8
9     public static void main(String [] args){
10         Test test= new Test();
11         MyTimer myTimer= new MyTimer(test,1000);
12         myTimer.start();
13         try{
14             Thread.sleep(7011);
15         }catch(InterruptedException e){
16             System.out.println(e);
17         }
18         myTimer.interrupt();
19     }
20
21     public void print(){
22         System.out.println("coucou");
23     }
24 }

```

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.

Exemple 5 : Testez la classe `TestDaemon` en modifiant l'état des commentaires dans le code.

```
1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Exemple de thread démon ou utilisateur
5  */
6 public class TestDaemon extends Thread {
7
8     public void run() {
9         for( int n=0; /*n < 200*/ ; ++n) {
10             System.out.println( "Boucle numéro "+n);
11         }
12     }
13
14     public static void main( String[] args) {
15         TestDaemon d ;
16         //d.setDaemon(true);
17         d = new TestDaemon();
18         //d.setDaemon(true);
19         d.start();
20         //d.setDaemon(true);
21     }
22 }
```

4 Les deux besoins de coordination de threads

4.1 Accès concurrents aux données

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.

La suite de ce TD s'attache à la résolution de ce problème par l'utilisation du mot clé `synchronised`. L'usage du mot clé `volatile` n'est pas abordé.

4.2 Concurrence et collaboration

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.

Les méthodes `wait()`, `notify()` et `join()` servent à faire face à ce type de situation. Elles ne sont pas analysées ici.

5 Synchronisation de l'accès aux données

5.1 Un exemple édifiant

Exemple 6 : Testez l'exemple suivant. Que constatez-vous ? Quelle en est la cause ?

```
1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Petite classe pourvue de deux méthodes simples
5  *
6  * Exemple inspiré par Thinking in Java, 3rd Edition, Beta
7  * Copyright (c)2002 by Bruce Eckel www.BruceEckel.com
8  */
9 public class ToujoursPair {
10
11     private int i = 0 ;
12
13     public void nextI(){
14         ++i ;
15         ++i ;
16     }
17
18     public int getI(){
19         return i ;
20     }
21 }
```

```
1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Thread accédant en lecture à une instance de ToujoursPair
5  *
6  * Exemple inspiré par Thinking in Java, 3rd Edition, Beta
7  * Copyright (c)2002 by Bruce Eckel www.BruceEckel.com
8  */
9 public class MyThread extends Thread{
10
11     ToujoursPair tp ;
12
13     public MyThread(ToujoursPair tp){
14         this.tp = tp ;
15     }
16
17     public void run(){
18         while(true){
19             int val = tp.getI() ;
20             if (val % 2 != 0){
21                 System.out.println("myThread : "+val) ;
22                 System.exit(0) ;
23             }
24         }
25     }
26 }
```

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Thread accédant en écriture et lecture à une instance
5  * de ToujoursPair
6  *
7  * Exemple inspiré par Thinking in Java, 3rd Edition, Beta
8  * Copyright (c)2002 by Bruce Eckel www.BruceEckel.com
9  */
10 public class Test {
11
12     public static void main(String [] args) {
13         ToujoursPair tp = new ToujoursPair () ;
14         MyThread t = new MyThread(tp) ;
15         t.start () ;
16         while(true){
17             tp.nextI () ;
18             if (tp.getI () % 1000000 == 0) {
19                 System.out.println(tp.getI ()) ;
20             }
21         }
22     }
23 }

```

5.2 Le mot clé synchronized

5.2.1 Modificateur de méthode d'instance

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 7 : 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**.

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Classe pourvue de deux méthodes d'affichage :
5  * illustration de l'utilisation du mot clé synchronized
6  * comme modificateur de méthode.
7  */
8 public class MyObject{
9
10    private String name ;
11
12    public MyObject(String name){
13        this.name = name ;
14    }
15
16    public void show(){
17        // public synchronized void show(){
18        String nom = Thread.currentThread().getName() ;
19        System.out.println("thread : "+nom+" | objet : "+name+" | in show");
20        try {
21            Thread.sleep(7000);
22        } catch(InterruptedException e){ }
23        System.out.println("thread : "+nom+" | objet : "+name+" | out show");
24    }
25
26    public void print(){
27        // public synchronized void print(){
28        String nom = Thread.currentThread().getName() ;
29        System.out.println("thread : "+nom+" | objet : "+name+" | in print");
30        try {
31            Thread.sleep(7000);
32        } catch(InterruptedException e){ }
33        System.out.println("thread : "+nom+" | objet : "+name+" | out print");
34    }
35 }

```

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Thread utilisant une méthode d'une instance de MyObject
5  */
6 public class MyThread extends Thread{
7
8    private MyObject myObject;
9
10    public MyThread(String name, MyObject myObject){
11        super(name) ;
12        this.myObject = myObject;
13    }
14    public void run(){
15        String nom = Thread.currentThread().getName() ;
16        System.out.println("thread : "+nom+" | in run");
17        myObject.show();
18        System.out.println("thread : "+nom+" | out run");
19    }

```

```

20 }

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Thread utilisant une méthode d'une instance de MyObject
5  * et instanciant une ou deux nouvelles threads.
6  */
7 public class Test{
8     public static void main(String [] args){
9         MyObject mol = new MyObject("mol") ;
10        //MyObject mo2 = new MyObject("mo2") ;
11        MyThread mt1 = new MyThread("mt1",mol) ;
12        //MyThread mt2 = new MyThread("mt2",mo2) ;
13
14        mt1.start();
15        //mt2.start();
16        try {
17            Thread.sleep(0L) ;
18        } catch (InterruptedException e){ }
19        mol.print() ;
20    }
21 }

```

5.2.2 Modificateur de bloc

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.

Exemple 8 : Testez l'exemple suivant en modifiant les objets de synchronisation des blocs `synchronized` de `fct` (quatre combinaisons différentes sont possibles).

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 import java.util.Random;
4
5 /**
6  * Classe pourvue d'une méthode d'affichage avec des blocs
7  * synchronized sur l'objet lui-même ou sur une chaîne de
8  * caractères.
9  */
10 public class MyObject{
11
12     private Random rnd ;
13
14     public MyObject(){
15         rnd = new Random() ;
16     }
17
18     public void fct(){
19         String nom = Thread.currentThread().getName() ;
20         System.out.println(nom+" : in fct") ;
21 }

```

```

22 synchronized(this) {
23     //synchronized("brol") {
24         System.out.println(nom+" : in bloc 1") ;
25         try {
26             Thread.sleep(rnd.nextInt(1000)) ;
27         } catch(InterruptedException e){ }
28         System.out.println(nom+" : out bloc 1") ;
29     }
30
31     System.out.println(nom+" : entre bloc 1 et bloc 2") ;
32     try {
33         Thread.sleep(rnd.nextInt(1000)) ;
34     } catch(InterruptedException e){ }
35
36     synchronized(this) {
37         //synchronized("brol") {
38             System.out.println( nom+" : in bloc 2") ;
39             try {
40                 Thread.sleep(rnd.nextInt(1000)) ;
41             } catch(InterruptedException e){ }
42             System.out.println(nom+" : out bloc 2") ;
43         }
44
45         System.out.println(nom+" : out fct") ;
46     }
47 }

```

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Thread utilisant une méthode d'une instance de MyObject.
5  */
6 public class MyThread extends Thread{
7
8     private MyObject myObject ;
9
10    public MyThread(String name, MyObject myObject){
11        super(name) ;
12        this.myObject = myObject ;
13    }
14    public void run(){
15        String nom = Thread.currentThread().getName() ;
16        System.out.println(nom+" : in run") ;
17        myObject.fct() ;
18        System.out.println(nom+" : out run") ;
19    }
20 }

```

```

1 package nvs.alg2ir.td04thread.enonce;
2
3 /**
4  * Classe de test instanciant deux threads utilisant une méthode à
5  * blocs synchronisés d'un même objet.
6  */
7 public class Test{

```

```

8  public static void main( String [] args) {
9
10     MyObject mo = new MyObject () ;
11     MyThread t1 = new MyThread(" t1" ,mo) ;
12     MyThread t2 = new MyThread(" t2" ,mo) ;
13
14     t1.start () ;
15     Thread.yield () ;
16     t2.start () ;
17 }
18 }

```

Lorsque la synchronisation se fait sur une chaîne de caractères, on parle de *synchronisation nommée*.

5.2.3 Appels imbriqués

Voyons maintenant ce qu'il advient de la clé lorsqu'une méthode `synchronized` est appelée depuis une autre également `synchronized` sur le *même* objet.

Exemple 9 : Reprenez les classes `Test` et `MyThread` de l'Exemple 8 pour tester la classe `MyObject` donnée ci-après en essayant les quatre combinaisons possibles de signatures des méthodes `fct` et `mtd` ou en ôtant la mise en commentaire du bloc synchronisé de `fct`.

```

1  package nvs.alg2ir.td04thread.enonce;
2
3  import java.util.Random;
4
5  /**
6   * Classe pourvue d'une méthode d'affichage synchronized
7   * appelant elle-même une méthode synchronized
8   */
9  public class MyObject {
10
11     private Random rnd ;
12
13     MyObject () {
14         rnd = new Random () ;
15     }
16
17     public void fct () {
18         // synchronized public void fct () {
19         //     synchronized (this) {
20             String nom = Thread.currentThread().getName () ;
21             System.out.println(nom+" : in fct" ) ;
22
23             try {
24                 Thread.sleep(rnd.nextInt(1000)) ;
25             } catch( InterruptedException e){ }
26
27             System.out.println(nom+" : goto mtd" ) ;
28             mtd () ;
29             System.out.println(nom+" : from mtd" ) ;

```

```
30
31     try {
32         Thread.sleep(rnd.nextInt(1000)) ;
33     } catch(InterruptedException e){ }
34
35     System.out.println(nom+" : out fct" ) ;
36 // } // fin du bloc synchronized
37 }
38
39 public void mtd() {
40 // public synchronized void mtd() {
41     String nom = Thread.currentThread().getName() ;
42     System.out.println(nom+" : in mtd" ) ;
43     try {
44         Thread.sleep(rnd.nextInt(2000)) ;
45     } catch(Exception e){ }
46     System.out.println(nom+" : out mtd" ) ;
47 }
48 }
```

Conclusion : la thread ne rend la clé que lorsqu'elle retourne de la méthode `synchronized` la plus externe. On parle de *verrouillage réentrant*.

5.2.4 Interblocage

Attention! Une remarque importante doit être faite. La plus grande prudence s'impose lors de l'appel d'une méthode `synchronized` à partir d'une méthode elle-même `synchronized`. Une *étrainte mortelle* (*interblocage*, *deadlock*) peut en effet survenir lorsque :

- 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 !

Ex2. Écrivez un code générant un interblocage.