

# le multithreading



Merci à Jean Michel Douin (douin@cnam.fr) pour son cours sur les threads dont certaines planches ont été reprises dans ce support.

Le cours de Jean Michel Douin est disponible en format .zip à [ftp://ftp.cnam.fr/pub4/CDL/java\\_c/Thread.zip](ftp://ftp.cnam.fr/pub4/CDL/java_c/Thread.zip)

## Définition

Une thread (appelée aussi processus léger ou activité) est une suite d'instructions à l'intérieur d'un process.

Les programmes qui utilisent plusieurs threads sont dits multithreadés.

## Syntaxe

Les threads peuvent être créés comme instance d'une classe dérivée de la classe `Thread`. Elles sont lancées par la méthode `start()`, qui demande à l'ordonnanceur de thread de lancer la méthode `run()` de la thread. Cette méthode `run()` doit être implantée dans le programme.

# Premier exemple

```
class DeuxThreadAsynchrones {
    public static void main(String args[ ]) {
        new UneThread("la thread 1").start();
        new UneThread("la seconde thread").start();
    }
}

class UneThread extends Thread {
    public UneThread(String str) {
        super(str);
    }
    public void run() {
        for (int i=0; i<10; i++) {
            System.out.println(i+" "+getName());
            try {sleep((int)(Math.random()*10));}
            catch (InterruptedException e){}
        }
        System.out.println(getName()+" est finie");
    }
}
```

**une exécution**

```
% java DeuxThreadAsynchrones
0 la thread 1
0 la seconde thread
1 la thread 1
2 la thread 1
1 la seconde thread
3 la thread 1
4 la thread 1
5 la thread 1
6 la thread 1
7 la thread 1
2 la seconde thread
3 la seconde thread
4 la seconde thread
8 la thread 1
5 la seconde thread
9 la thread 1
6 la seconde thread
la thread 1 est finie
7 la seconde thread
8 la seconde thread
9 la seconde thread
la seconde thread est finie
```

# Une classe "threadée"

C'est une classe qui implémente une thread. Syntactiquement on la construit :

- ou bien comme classe dérivée de la classe Thread. Par exemple

```
class MaClasseThread extends Thread {  
    ...  
}
```

- ou bien comme implémentation de l'interface Runnable.

```
class MaClasseThread implements Runnable {  
    ...  
}
```

Cette dernière solution est la seule possible pour une applet "threadée" puisque Java ne supporte pas l'héritage multiple :

```
public class MonAppletThread extends Applet implements  
Runnable {  
    ...  
}
```

**[www.Mcours.com](http://www.Mcours.com)**  
Site N°1 des Cours et Exercices Email: [contact@mcours.com](mailto:contact@mcours.com)

# constructeurs de thread

Il y en a plusieurs. Quand on écrit

```
t1 = new MaClasseThread();
```

le code lancé par `t1.start()` ; est le code `run()` de la classe threadée `MaClasseThread`.

Si on écrit :

```
t1 = new Thread(monRunnable);
```

le code lancé par `t1.start()` ; est le code `run()` de la classe `monRunnable`. En général on a cette seconde syntaxe pour les applets qui implantent la méthode `run()` et la première syntaxe pour les applications Java.

Exemple :

```
class Appli extends Thread {
    Thread t1;
    public static void main(String args[]) {
        t1 = new Appli();
        t1.start();
    }
    public void run() { ...
        // ce code est lancé
    }
}
```

et

```
class MonApplet extends Applet implements Runnable {
    public void init() {
        Thread t1 = new Thread(this);
        t1.start();
    }
    public void run() { ...
        // ce code est lancé
    }
}
```

# le multithreading dans les applets

Il est bien de prévoir dans une applet, l'arrêt de traitement "désagréable" (animation ou musique intempestive) par clics souris. Il suffit décrire :

```
boolean threadsuspendue = false;

public boolean mouseDown(Event e, int x, int y) {
    if (threadsuspendue) {
        lathread.resume();
    } else {
        lathread.suspend();
    }
    threadsuspendue = !threadsuspendue;
    return true;
}
```

# le multithreading dans les applets (suite)

## Méthodes `stop()` et `start()`

Quand un browser est iconifié ou qu'il change de page, la méthode `stop()` de l'applet est lancée. Celle ci doit libérer les ressources entre autre le processeur. Si aucune thread n'a été implantée dans l'applet, le processeur est libéré. Sinon il faut (sauf bonne raison) arrêter les threads en écrivant :

```
public void stop() {
    if (lathread != null) {
        lathread.stop();
        lathread = null;
    }
}
```

Pour relancer une thread arrêtée par `stop()`, il faut implanter la méthode `start()` par :

```
public void start() {
    if (lathread == null) {
        lathread = new Thread (...);
        lathread.start(); // ce N'est Pas recursif. OK ?
    }
}
```

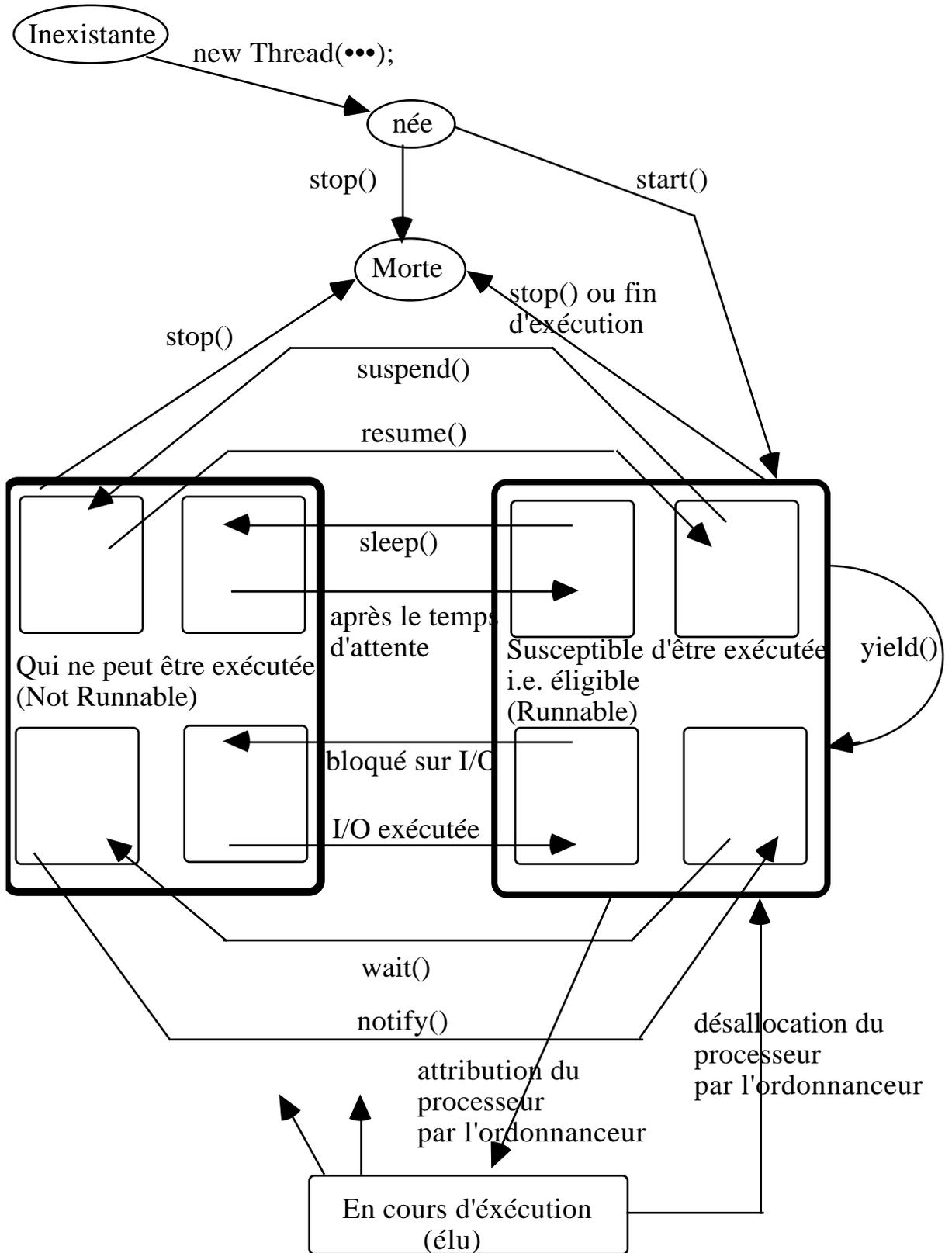
# Passage de paramètres

```
class AutoThreadParam implements Runnable{
    private Thread local;
    private String param;

    public AutoThreadParam(String param){
        this.param = param;
        local = new Thread( this);
        local.start();
    }
    public void run(){
        if( local == Thread.currentThread()){
            while(true){
                System.out.println("dans
AutoThreadParam.run"+ param);
            }
        }
    }
}

public class Thread4 {
    public static void main(String args[]) {
        AutoThreadParam auto1 = new
AutoThreadParam("auto 1");
        AutoThreadParam auto2 = new
AutoThreadParam ("auto 2");
        while(true){
            System.out.println("dans Thread4.main");
        }
    }
}
```

# Les états d'une thread



# Les états d'une thread (suite)

À l'état "née", une thread n'est qu'un objet vide et aucune ressource système ne lui a été allouée.

On passe de l'état "née" à l'état "Runnable" en lançant la méthode `start()`. Le système lui alloue des ressources, l'indique à l'ordonnanceur qui lancera l'exécution de la méthode `run()`.

A tout instant c'est une thread éligible de plus haute priorité qui est en cours d'exécution.

# Les états d'une thread (suite)

On entre dans l'état "Not Runnable" suivant 4 cas :

- quelqu'un a lancé la méthode `suspend()`
- quelqu'un a lancé la méthode `sleep()`
- la thread a lancé la méthode `wait()` en attente qu'une condition se réalise.
- la thread est en attente d'entrées/sorties

Pour chacune de ces conditions on revient dans l'état "Runnable" par une action spécifique :

- on revient de `sleep()` lorsque le temps d'attente est écoulé
- on revient de `suspend()` par `resume()`
- on revient de `wait()` par `notify()` ou `notifyAll()`.
- bloqué sur une E/S on revient a "Runnable" lorsque l'opération d'E/S est réalisé.

On arrive dans l'état "Morte" lorsque l'exécution de `run()` est terminée ou bien après un appel à `stop()`.

# méthodes publiques de "Thread"

```
final void suspend();  
final void resume();  
static native void yield();  
final native boolean isAlive();  
final void setName(String Nom);  
final String getName();  
public void run();
```

# Ordonnancement des threads

A toute thread est associée une priorité. Priorité de 1 (`Thread.MIN_PRIORITY`) à 10 (`Thread.MAX_PRIORITY`), par défaut 5. Une thread adopte la priorité de son processus créateur. `setPriority(int p)` permet de changer celle-ci

Ordonnancement dépendant des plateformes :

Sous Windows 95 (JDK 1.1.4), et MacOS (JDK 1.0.2), un tourniquet pour les threads de même priorité est installé.

- Le choix de la thread active parmi les éligibles de même priorité est arbitraire, Sous Solaris de version inférieure à 2.5 (JDK 1.0.2 et 1.1), pas de tourniquet (sur les threads de même priorité).

La sémantique de la méthode `yield()` n'est pas définie, certaines plate-formes peuvent l'ignorer.

A priorité égale : la spécification n'impose rien

- Pas de soucis d'équité  
- Après un `notify()` c'est l'un des processus en attente qui est choisi.

# Ordonnancement des threads (suite)

```
class ThreadExtends extends Thread {
    public void run(){
        while(true){
            System.out.println("dans
ThreadExtends.run");
        }
    }
}

public class Thread1 {

    public static void main(String args[]) {
        ThreadExtends t0 = new ThreadExtends();
        t0.start();
        while(true){
            System.out.println("dans Thread1.main");
        }
    }
}
```

[www.Mcours.com](http://www.Mcours.com)

Site N°1 des Cours et Exercices Email: [contact@mcours.com](mailto:contact@mcours.com)

**trace de l'exécution****sous Windows 95 :**

```
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans ThreadExtends.run
dans ThreadExtends.run
dans Thread1.main
dans Thread1.main
dans ThreadExtends.run
dans ThreadExtends.run
dans ThreadExtends.run
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans Thread1.main
dans ThreadExtends.run
```

**==> Ordonnanceur de type tourniquet sous Windows 95**

**sous Solaris < 2.6 :**

```
% java Thread1
dans Thread1.main
```

• • •

**==> monopolisation du processeur sous Solaris : thread égoïste (selfish thread)**

## So we can say

"At any given time, when multiple threads are ready to be executed, the thread with the highest priority is chosen for execution. Only when that thread stops, or is suspended for some reason, will a lower priority thread start executing.

Scheduling of the CPU is fully preemptive. If a thread with a higher priority than the currently executing thread needs to execute, the higher priority thread is immediately scheduled.

The Java runtime will not preempt the currently running thread for another thread of the same priority. In other words, the Java runtime does not time-slice. However, the system implementation of threads underlying the Java Thread class may support time-slicing."

source : The Java Tutorial

# Concurrence, synchronisation en Java

usage du mot-clé synchronized

Utilisation de synchronized

1er cas :

```

|| synchronized(obj){
||     // ici le code atomique sur l'objet obj
|| }
||

```

2ime cas

```

|| synchronized void p(){ .....}
||

```

qui signifie :

```

|| void p(){
||     synchronized (this){.....}
|| }
||

```

Dans java.lang.Object

```

final void wait() throws
InterruptedException
final native void wait(long timeout)
throws InterruptedException

```

...

**Notifications**

```

final native void notify()
final native void notifyAll()

```

# `wait()`, `notify()` de la classe `Object`

## remarque fondamentale

Ce sont des méthodes de la classe `Object` pas de la classe `Thread` i.e. elle implémentent des mécanismes de demande de verrou et d'avertissement pas d'attente (i.e. `wait()` de `Object` `wait()` de `Thread`).

`notify()` (respectivement `notifyAll()`) avertit une des threads qui attend (respectivement toutes les threads qui attendent) sur un `wait()` sur l'objet sur lequel a été lancé `notify()` et débloque le `wait()` sur cet objet. Il faut donc avant de faire un `notify()` ou `notifyAll()` changer la condition de boucle qui mène au `wait()`.

"The `notifyAll` method wakes up all threads waiting on the object."

La méthode `wait()` relache le moniteur de l'objet sur lequel elle a été lancée (sinon tout serait bloqué !!).

Ces deux méthodes doivent être lancées dans des méthodes `synchronized`.

# Producteur- consommateur : un exemple

Un producteur est une thread qui dépose des jetons numérotés dans un chapeau qui ne peut contenir qu'un seul jeton. Un consommateur prend ce jeton qui doit être présent dans le chapeau. Donc :

- le producteur doit s'arrêter de déposer des jetons lorsqu'il y en a déjà un et doit être informer qu'un jeton a été retiré.
- le consommateur ne peut pas prendre de jeton s'il n'y en a pas (arrêt du consommateur) et doit être informé lorsqu'un jeton a été déposé.

L'objet le plus à même pour avertir le producteur et le consommateur est le chapeau lui même.



## fichier Producer.java

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" +
this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

## fichier Consumer.java

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" +
this.number + " got: " + value);
        }
    }
}
```

## fichier CubbyHole.java

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }

    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}
```

## fichier ProducerConsumerTest.java

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
```

# Sorties du programmes :

```
Producer #1 put : 0
Consumer #1 got : 0
Producer #1 put : 1
Consumer #1 got : 1
Producer #1 put : 2
Consumer #1 got : 2
Producer #1 put : 3
Consumer #1 got : 3
Producer #1 put : 4
Consumer #1 got : 4
Producer #1 put : 5
Consumer #1 got : 5
Producer #1 put : 6
Consumer #1 got : 6
Producer #1 put : 7
Consumer #1 got : 7
Producer #1 put : 8
Consumer #1 got : 8
Producer #1 put : 9
Consumer #1 got : 9
```

# Remarques

Avec ce même programme, il peut y avoir plusieurs producteurs et plusieurs consommateurs (repérés par des numéros) mais toujours un seul chapeau ayant un emplacement pour un seul jeton.

Comme la notification est faite par un `get()` d'un consommateur ou un `put()` d'un producteur, on doit vérifier si c'est un producteur ou un consommateur qui a levé l'arrêt ce qui est fait dans le test et qui doit être refait à chaque déblocage du `wait()` d'où la nécessité d'une boucle `while` et non d'un test `if`.

# Exemple avec 3 producteurs et 3 consommateurs.

Le programme "lanceur" devient :

fichier `ProducerConsumerTest2.java`

```
public class ProducerConsumerTest2 {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Producer p2 = new Producer(c, 2);
        Producer p3 = new Producer(c, 3);
        Consumer c1 = new Consumer(c, 1);
        Consumer c2 = new Consumer(c, 2);
        Consumer c3 = new Consumer(c, 3);
        p1.start(); p2.start(); p3.start();
        c1.start(); c2.start(); c3.start();
    }
}
```



---

## Exécution

```
% java ProducerConsumerTest2
Producer #1 put: 0
Consumer #1 got: 0
Producer #2 put: 0
Consumer #1 got: 0
Producer #3 put: 0
Consumer #1 got: 0
Producer #3 put: 1
Consumer #2 got: 1
Producer #1 put: 1
Consumer #3 got: 1
Producer #3 put: 2
Consumer #1 got: 2
Producer #2 put: 1
Consumer #2 got: 1
Producer #3 put: 3
Consumer #3 got: 3
Producer #3 put: 4
Consumer #1 got: 4
Producer #1 put: 2
Consumer #2 got: 2
Producer #3 put: 5
Consumer #3 got: 5
Producer #2 put: 2
Consumer #1 got: 2
Producer #2 put: 3
Consumer #2 got: 3
Producer #1 put: 3
Consumer #3 got: 3
Producer #3 put: 6
Consumer #1 got: 6
Producer #1 put: 4
Consumer #1 got: 4
Producer #3 put: 7
Consumer #2 got: 7
Producer #2 put: 4
Consumer #3 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #2 got: 6
Producer #3 put: 8
Consumer #3 got: 8
Producer #2 put: 5
Consumer #1 got: 5
Producer #1 put: 7
...

```

# Le mot clé réservé synchronized

utilisé comme modifieur de méthode de classe ou d'instance (cf. ci dessus) :

```
|| public synchronized void methAtomique() { ... } ||
```

ou

```
|| public static synchronized void methClasseAtomique() { ... } ||
```

synchronized peut aussi être utilisé pour un bloc d'instructions :

```
|| synchronized (expression) { ... } ||
```

où expression repère un objet.

## **synchronized : modifieur de méthode d'instance**

Lorsqu'une thread veut lancer une méthode d'instance `synchronized`, le système Java pose un verrou sur l'instance. Par la suite, une autre thread invoquant une méthode `synchronized` sur cet objet sera bloquée jusqu'à ce que le verrou soit levé.

On implante ainsi l'exclusion mutuelle entre méthode `synchronized`. Attention les méthodes non `synchronized` ne sont pas en exclusion mutuelle comme le montre l'exemple (Henri ANDRÉ) :

---

## synchronized : modifieur de méthode d'instance (suite)

```
public class T09Synchro {
    T09Object obj;

    public static void main( String argv[] ){
        new T09Synchro().go();
        System.out.println("");
    }

    public void go() {
        obj = new T09Object();

        Runnable r = new T09Accede( obj );
        Thread a = new Thread( r, "a" );
        Thread b = new Thread( r, "b" );
        Thread c = new Thread( r, "c" );
        a.start();
        b.start();
        c.start();
    }
}

class T09Accede implements Runnable{
    T09Object  obj;
    T09Accede( T09Object obj ) {
        this.obj = obj;
    }
    public void run() {
        if (Thread.currentThread().getName().equals( "a"
        )) obj.m1();
        if (Thread.currentThread().getName().equals( "b"
        )) obj.m2();
        if (Thread.currentThread().getName().equals(
        "c")) obj.m3(); }
}
```

```
class T09Object {
public synchronized void m1() {
    for ( int i = 0; i < 10; i++ ) {
        System.out.print("1");
        try { Thread.sleep( 1000 );}
        catch(InterruptedExpection e){}
    }
}

public synchronized void m2() {
    for ( int i = 0; i < 10; i++ ) {
        System.out.print("2");
        try { Thread.sleep( 1000 );}
        catch(InterruptedExpection e){}
    }
}

public void m3() {
    for ( int i = 0; i < 10; i++ ) {
        System.out.print("3");
        try { Thread.sleep( 1000 );}
        catch(InterruptedExpection e){}
    }
}
}
```

donne comme sorties :

13131313131313131313132222222222

De même si une thread veut lancer une méthode de classe (i.e. static) synchronized, le système Java pose un verrou et aucune autre thread ne pourra lancer une méthode de classe synchronized.

# synchronized pour un bloc

Dans

```
|| synchronized (expression) { ... } ||
```

où expression repère un objet, le système Java pose un verrou sur cet objet pendant l'exécution du bloc. Aucun autre bloc `synchronized` sur cet objet ne peut être exécuté et un tel bloc ne peut être lancé qu'après avoir obtenu ce verrou.

On définit ainsi une section critique  
exemple :

```
|| public void trie(int[] tableau) { ||  
||     synchronized (tableau) { ||  
||         // votre algorithme de tri préféré ||  
||     } ||  
|| } ||
```

remarque

En fait

```
|| synchronized void meth() {...} ||
```

signifie

```
|| void meth() {synchronized (this) {...}} ||
```

# Bibliographie

Java Threads Scott Oaks & Henry Wong.  
O'Reilly.

The Java Tutorial. Maria Campione, Kathy  
Walrath. Java Series. Addison Wesley

Tha Java Programming language. Ken  
Arnold, James Gosling. Addison Wesley

Monitors, an operating system structuring  
concept. C.A.R. Hoare. CACM, vol 17, n° 10,  
Oct 1974. pages 549-557.

Communicating Sequential Processes. C.A.R.  
Hoare. CACM, vol 21, n°8, Aug.1978. pages  
666-677

cours Jean Michel Douin en format .zip à  
[ftp://ftp.cnam.fr/pub4/CDL/java\\_c/Thread.zip](ftp://ftp.cnam.fr/pub4/CDL/java_c/Thread.zip)

**le cours en ligne à :**

<http://www.javasoft.com/docs/books/tutorial/essential/threads/definition.html>