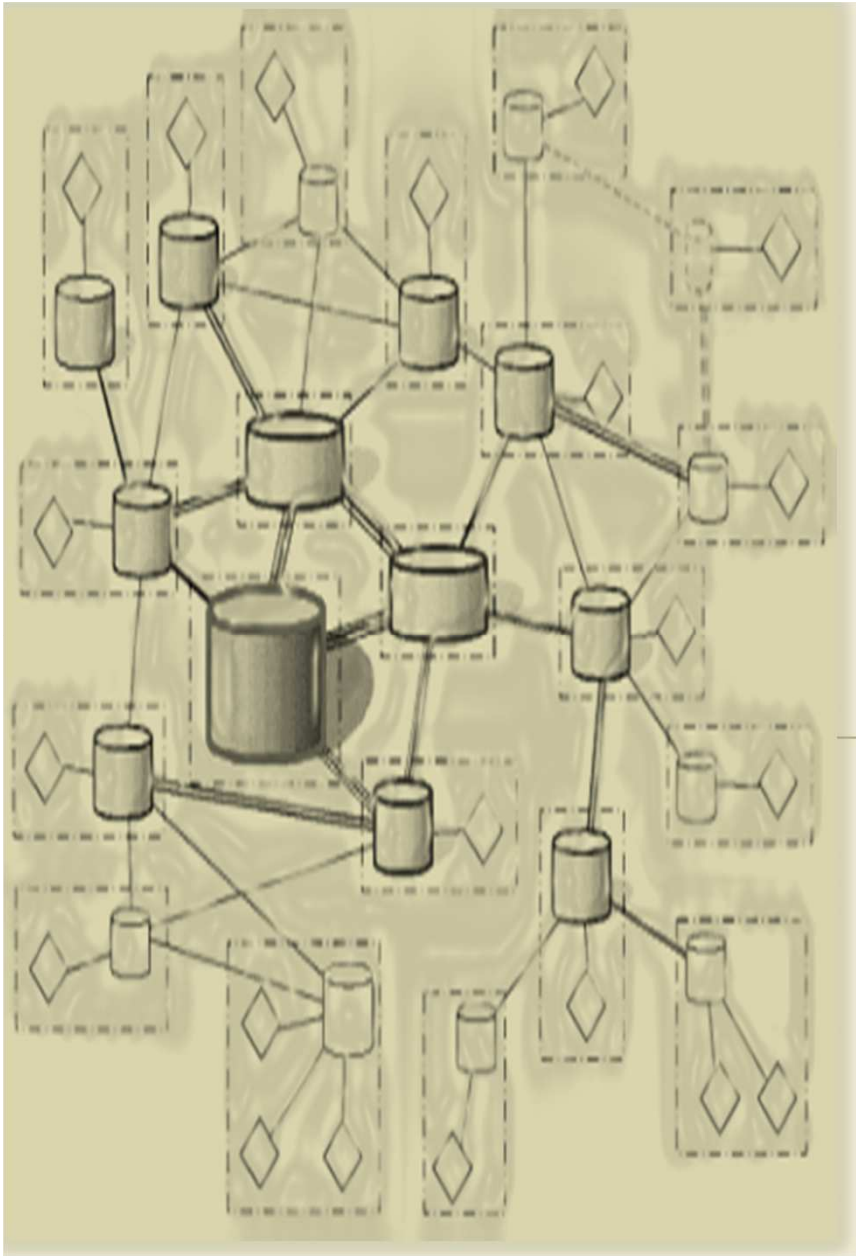


DÉVELOPPEMENT D'APPLICATIONS RÉPARTIES

Chapitre 3
Les Threads en Java

©Amen Ben Hadj Ali

amenbha@hotmail.com

ISI-L3SIL

2011-2012

Chapitre 3- Les Threads en Java

Plan

Introduction

Définition

- Les threads (tâches ou fil d'exécution) permettent le déroulement de plusieurs traitements de façon simultanée.
- On les appelle processus légers car ils sont internes à une même application et partagent donc le même espace d'adressage mémoire alors que les processus lourds sont gérés par le système d'exploitation.

- → Plusieurs threads peuvent s'exécuter en parallèle dans une application
- Problème : Synchronisation inter-thread

Introduction

Exécution

- Au niveau matériel, un seul thread peut occuper le processeur, l'exécution des autres threads est alors suspendue. C'est ce que l'on appelle l'accès concurrent (ils se partagent l'accès processeur).

- Une fois le temps processeur attribué au thread actif écoulé, celui-ci est alors suspendu et un autre thread est exécuté ou réactivé par le processeur.

Introduction

Exécution

- Un thread actif peut être suspendu même s'il n'a pas fini son exécution.

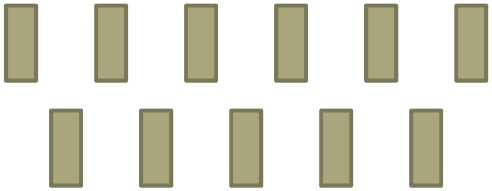
Au niveau de l'application

Thread 1

Thread 2

On a l'illusion que tous les threads s'exécutent en parallèle

Au niveau du processeur



En réalité, les threads sont exécutés en alternance: le système se charge d'allouer régulièrement une tranche de temps pour qu'il puisse exécuter les threads

Introduction

Utilité des threads

Les threads permettent de :

- Créer des processus séparés, « indépendants » qui exécuteront chacun une tâche précise sans bloquer le déroulement du programme.
- Accéder à la programmation concurrente (parallèle) pour que plusieurs threads puissent travailler sur les mêmes données.
- Gérer des interfaces utilisateurs sophistiquées et la gestion de graphiques animés.
- Exploiter au mieux les ordinateurs dotés de plusieurs processeurs.

Chapitre 3- Les Threads en Java

Plan

Exécution d'un thread

Cycle de vie d'un thread

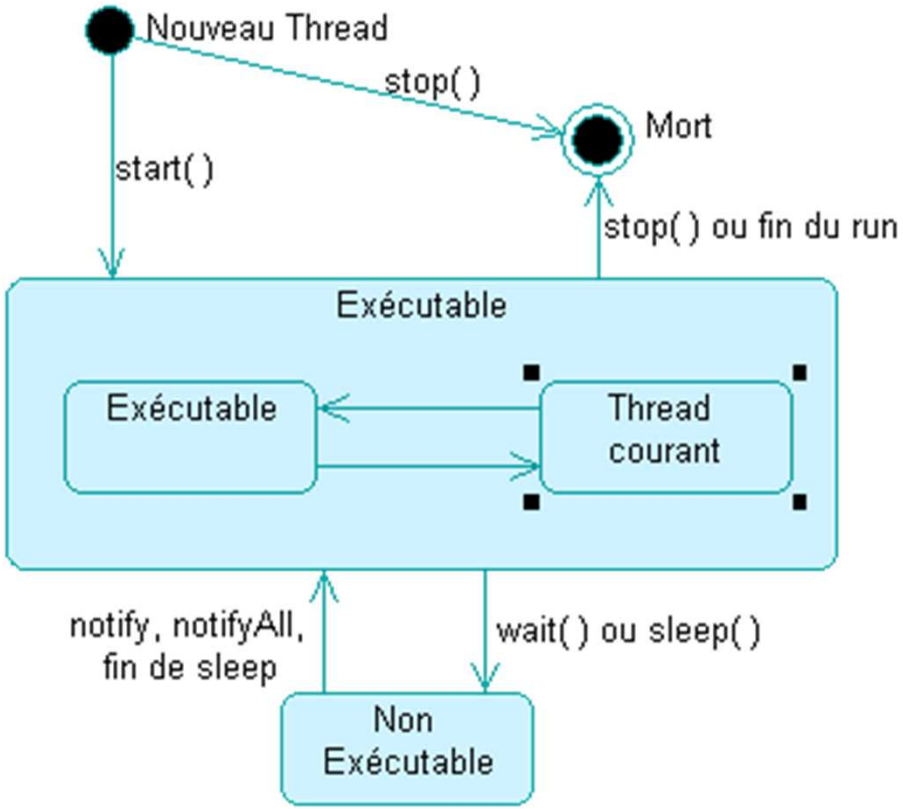


Diagramme illustrant les différents états ainsi que les différentes transitions possibles d'un cycle de vie d'un Thread.

Exécution d'un thread

Démarrage d'un thread : start() et run()

En Java, il existe une classe *Thread* qui contient plusieurs méthodes qui permettent le threading d'une application. C'est-à-dire pouvoir faire exécuter plusieurs traitements en parallèle ou encore utiliser des boucles infinies sans bloquer le bon déroulement du programme. Il suffit de faire hériter la classe que l'on désire « threader » de la classe **Thread (*java.lang.Thread)***.

Exécution d'un thread

- La classe ***java.lang.Thread*** modélise un thread :
  - **currentThread()** : donne le thread actuellement en cours d'exécution
  - **setName()** : fixe le nom du thread
  - **getName()** : nom du thread
  - **isAlive()** : indique si le thread est actif(true) ou non (false)
  - **start()** : lance l'exécution d'un thread
  - **run()** : méthode exécutée automatiquement après que la méthode start précédente ait été exécutée
  - **sleep(n)** : arrête l'exécution d'un thread pendant n millisecondes
  - **join()** : opération bloquante - attend la fin du thread pour passer à l'instruction suivante
- Les constructeurs sont :
  - **Thread()** : crée une référence sur une tâche asynchrone. Celle-ci est encore inactive. La tâche créée doit posséder la méthode **run** : ce sera le plus souvent une classe dérivée de **Thread** qui sera utilisée.
  - **Thread(Runnable object)** : idem mais c'est l'objet **Runnable** passé en paramètre qui implémente la méthode **run**.

Exécution d'un thread

Public void start(): alloue les ressources nécessaires à l'exécution d'un thread et invoque la méthode run()

Req: si vous invoquez la méthode run (au lieu de start) le code s'exécute bien, mais aucun nouveau thread n'est lancé dans le système. Inversement, la méthode start, lance un nouveau thread dans le système dont le code à exécuter démarre par le run.

Public void run() { }

- toujours de type public,
- elle est héritée de la classe Thread
- doit être redéfinie si vous désirez que votre thread fasse quelque chose.
- contient tout le code qui va être exécuté par le thread.
- lancée par la méthode start().
- Tant que la méthode start() n'est pas lancée, le bloc de code de la méthode run() n'est pas exécuté.

Exécution d'un thread

Démarrage d'un thread : start() et run()

```
// premier test
public class TestThread extends Thread {
   ...
 public TestThread() {  }

public void run() {

public static void main(String[] args) {
            TestThread t1 = new TestThread();
            t1.start();
 } }
```

Lorsque le thread a fini d'exécuter la méthode ***run()*** il entre dans un état **dead** et ne peut pas être relancé par la méthode ***start()***. Cependant, l'instance est toujours présente ce qui permet d'avoir des informations sur l'état du thread.

Exécution d'un thread

Suspension d'un Thread: Sleep (long) et wait()

- **La méthode statique *sleep(long)*:**
  - Elle veut dire patienter ou endormir pendant un certain temps, va permettre de suspendre l'exécution d'un thread.
  - Elle prend comme paramètre un temps en milliseconde (un **long**) qui correspond au temps d'attente avant de poursuivre le déroulement du thread.
  - Dans le cas d'une animation graphique, pour éviter un affichage trop rapide, on utilise cette méthode.
  - Elle lance l'exception **InterruptedException** si le Thread est stoppé pendant son sommeil (par un appel de la méthode ***interrupt()*** par exemple)

Exécution d'un thread

Suspension d'un Thread: Sleep (long) et wait()

- **La méthode wait():**
  - La méthode ***wait()*** provoque la suspension de l'exécution du thread dans lequel s'exécutait le code courant.
  - ***wait(long):*** spécifier une durée
  - Un appel aux méthodes ***notify()*** ou ***notifyAll()*** permet au **Thread** de sortir de cet état d'attente. S'il y a plusieurs threads en attente, c'est celui qui a été suspendu le plus longtemps qui est réveillé.
  - La méthode wait suspend l'exécution d'un thread, en attendant qu'une certaine condition soit réalisée. La réalisation de cette condition est signalée par un autre thread par les méthodes notify ou notifyAll.
  - Wait, notify et notifyAll sont définies dans la classe java.lang.Object et sont donc héritées par toute classe

Exécution d'un thread

- **La méthode join():**
- La méthode ***join()*** fait attendre la fin d'un thread par le thread en cours d'exécution. Elle permet de s'assurer que le traitement du thread est terminé avant de poursuivre le déroulement du programme
- ***join(long) ou join(long, int) : millisecondes et nanosecondes***

```
Thread {
            run() {
            System.out.println("Début du long traitement");
           sleep(3000);
           System.out.println("Fin du long traitement");
           } (InterruptedException e) { e.printStackTrace(); } }
main(String[] args) {
            {
             Thread t = MyThread();
              t.start();
              System.out.println("On attend la fin du thread");
               t.join();
               System.out.println("Le thread a fini son exécution");
          }(InterruptedException e) { e.printStackTrace(); } }
```

InterruptedException est levée dans les cas suivants:

1. Un Thread t1 n'a pas fini son exécution
2. Un Thread t2 appelle t1.join()
3. t2.interrupt() est appelée

Exécution d'un thread (exemple)


```
// premier test avec la classe thread
public class FirstThread extends Thread {
   private String threadName;
   public FirstThread(String threadName) {
        this.threadName = threadName;
        this.start();    }
    public void run() {
         try {  for(int i=0;i<5;i++) {
                        System.out.println( "Thread: " + this.threadName + " -  " + i );
                        Thread.sleep(30); // sleep peut déclencher une exception
        } } catch (InterruptedException exc) {
                              exc.printStackTrace();        }    }
 public static void main(String[] args) {
                FirstThread thr1 = new FirstThread("Toto");
} }
```

L'exécution :

Thread: toto 0
Thread: toto 1
Thread: toto 2
Thread: toto 3
Thread: toto 4

Exécution d'un thread

Tester l'exécution d'un Thread: isAlive()

- La méthode ***isAlive()*** permet de savoir si un thread est toujours en vie.
- On considère qu'un thread est vivant s'il a démarré (la méthode ***start()*** a été appelée) et qu'il n'est encore terminé (il n'est pas sorti de sa méthode ***run()***).
- Cette méthode retourne un booléen : **true** si le thread est dans sa méthode ***run()***, **false** dans les autres cas.

```
Boolean etat;
Thread t1 = Thread();
etat = t1.isAlive
```

Exécution d'un thread

Arrêt d'un Thread

- Pour provoquer l'arrêt d'un thread, il est conseillé de faire une vérification périodique d'une variable de celui-ci : il peut s'agir d'un **boolean** que vous mettrez à **true** lorsqu'il est nécessaire de stopper le thread. Ainsi, en testant et prévoyant proprement cette condition, vous pouvez libérer proprement les ressources utilisées par le thread ainsi que les verrous posés et terminer l'exécution de la méthode ***run()***.

- Si votre thread procède à des attentes (à l'aide de ***wait()***, ***join()*** ou ***sleep()***), il ne sera pas toujours possible de procéder de cette manière. Dans ces cas-là, vous pouvez avoir recours à la méthode ***interrupt()*** qui provoque la levée d'une exception **InterruptedException** ce qui vous permettra de libérer proprement les ressources dans un bloc **catch** avant de terminer la méthode ***run()***.

Exécution d'un thread

Arrêt d'un Thread

- De même, si votre thread est bloqué sur une opération d'entrées/sorties sur un canal interruptible (**java.nio.channel.InterruptibleChannel**), il est aussi possible d'utiliser la méthode ***interrupt()*** qui fermera celui-ci et lèvera une exception **ClosedByInterruptException**.

- Il existe une méthode ***stop()*** qui n'a été conservée que pour des questions de compatibilité mais celle-ci est fortement **déconseillée** (***deprecated***) : en effet celle-ci provoque l'arrêt brutal du thread et ne permet pas de vérifier que toutes les opérations de libération des ressources ou verrous s'effectuent correctement.

Exécution d'un thread

Priorité des threads

- Sur une durée déterminée, un thread ayant une priorité plus haute recevra plus fréquemment le processeur qu'un autre thread.
- La priorité d'un thread est définie par un nombre variant entre 1 et 10 (10 correspond à la priorité maximale).
- La classe **Thread** possède trois variables statiques de type **int** correspondants à des priorités différentes que le programmeur pourra affecter aux threads qu'il créera.
- **MIN_PRIORITY** : correspond à la priorité minimale.
  MAX_PRIORITY : correspond à la priorité maximale.
  NORM_PRIORITY : correspond à la priorité par défaut.
- On peut changer la priorité d'un thread grâce à la méthode ***setPriority(int)***. On peut aussi récupérer la priorité d'un thread par l'intermédiaire de la méthode ***getPriority()***, cette méthode retourne un **int**.

```
Thread t = Thread();
t.setPriority(Thread.MAX_PRIORITY - 2);
System.out.println(t.getPriority());
```

L'interface Runnable

Création d'un thread

- Il existe une autre façon de « threader » une classe, qui consiste à implémenter l'interface Runnable. La classe **Thread** implémente l'interface **Runnable** avec une méthode run() vide
- L'intérêt d'utiliser l'interface **Runnable** plutôt que la classe **Thread**, est qu'elle permet l'héritage d'une autre classe, chose impossible avec la classe **Thread** puisque Java ne permet pas l'héritage multiple.
- Implémenter l'interface **Runnable** offre aussi l'avantage de pouvoir partager des données entre threads sans avoir recours au mot clé **static.**
- L'exécution d'un thread qui implémente l'interface **Runnable** diffère un peu de ce que nous avons vu précédemment avec un thread qui hérite de la classe **Thread.**

L'interface Runnable

Exemple

Nous devons pour cela créer une instance de la classe **Thread** avec comme paramètre l'instance de la classe qui implémente l'interface Runnable :

```
// Second test de classe de thread
 public class SecondThread implements Runnable {
    private int counter;
  public SecondThread (int counter) {
       this.counter = counter;
       for (int i=0;i<5;i++) {
            (new Thread(this)).start();        }    }
  public void run() {
  try {   for(int i=0;i<50;i++) {
                System.out.println( "Valeur du compteur == " + counter++  );
               Thread.sleep(10);
           } } catch (InterruptedException exception) {
exception.printStackTrace();        }    }
static public void main(String argv[]) {
     SecondThread p1 = new SecondThread(0);
//SecondThread p2= new SecondThread(0); }}
```

L'interface Runnable

CurrentThread

Lorsque vous implémentez l'interface **Runnable**, vous n'avez plus directement accès aux méthodes de la classe **Thread** pour contrôler le thread en cours d'exécution. En effet, l'objet passé en paramètre du constructeur ***Thread()*** n'est pas le **Thread** qui sera exécuté mais servira juste de base pour celui-ci. Pour y avoir accès, vous pouvez utiliser la méthode statique ***currentThread()*** de la classe **Thread** qui retourne le thread en cours d'exécution, ce qui vous permettra de continuer à utiliser les méthodes de la classe **Thread** sur celui-ci.

Voici un exemple avec la méthode ***getName()*** de la classe **Thread** qui renvoie le nom du thread :

```
Class curent implements Runnable {
    public void run() {
        System.out.println("Mon nom est " +
        Thread.currentThread().getName()); }

main(String[] args) {
        (new Thread(new curent)).start(); } }
```

On obtient le même affichage que ce programme:

```
MyThread {
        public void run() {
 System.out.println("Mon nom
   est " + getName()); }

main(String[] args) {
 { MyThread().start();}}
```

L'interface Runnable

Résumé

	Avantages	Inconvénients
extends java.lang.Thread	Chaque thread a ses données qui lui sont propres.	On ne peut plus hériter d'une autre classe.
implements java.lang.Runnable	L'héritage reste possible. En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs de votre classe sont partagés pour tous les threads qui y sont basés. Dans certains cas, il peut s'avérer que cela soit un atout.

Partage d'informations entre threads

Les variables statiques

- Il existe deux manières de partager les données entre threads, la manière la plus simple étant les variables statiques, il est aussi possible de créer des objets **Thread** à partir d'un seul et même objet afin que seuls ces objets partagent des informations en commun.
- Une variable statique est une variable de classe, c'est-à-dire qu'elle est partagée par toutes les instances d'une classe.

```
MyThread {
            static int a=0; // la variable est partagée
            public void run() { i = 0; i < 5; i++) {
            System.out.println(getName() + ": " + a++); } }
   main(String[] args) {
            MyThread().start();
            MyThread().start();
} }
```

```
Thread-1: 0
Thread-0: 1
Thread-1: 2
Thread-0: 3
Thread-1: 4
Thread-0: 5
Thread-1: 6
Thread-0: 7
Thread-1: 8
Thread-0: 9
```

Partage d'informations entre threads

Constructeurs Threads (Runnable)

Utiliser des variables statiques apporte une grande facilité d'implémentation des accès partagés. Cependant cette solution peut ne pas correspondre à vos besoins dans certains cas où vos classes ne sont pas destinées qu'à être utilisées en tant que threads concurrents. Ainsi, il peut être nécessaire de ne faire partager des données qu'entre threads 2-à-2 (ou plus), sans pour autant que tous les threads ne partagent les même variables ; le constructeur de la classe **Thread** permet d'implémenter cette solution en proposant d'utiliser un même objet pour créer plusieurs threads.

```
Class MyRunnable implements Runnable{
   int a=0, b=0; String text;
   MyRunnable(String text) { this.text = text; }
    public void run() {
              for(int i = 0; i < 3; i++) {
               System.out.println(text + " " + Thread.currentThread().getName() + " a: " + a++);
               System.out.println(text + " " + Thread.currentThread().getName() + " b: " + b++); } }
main(String[] args) {
              MyRunnable o1= MyRunnable("Groupe 1");
              Thread(o1).start();
               MyRunnable o2 = MyRunnable("Groupe 2");
              Thread(o2).start();}
```

```
Groupe 1 Thread-0 a: 0
Groupe 1 Thread-1 a: 1
Groupe 2 Thread-3 a: 0
Groupe 1 Thread-0 b: 0
Groupe 2 Thread-2 a: 1
Groupe 1 Thread-1 b: 1
```

Partage d'informations entre threads

Constructeurs Threads (Runnable)

- Ainsi l'utilisation de **membres statiques** permet de spécifier quelles variables membres l'ont désire partager ; les autres membres ne seront pas partagés. L'inconvénient étant que tous les objets issus de la classe partageront ces données, même des objets nouvellement créés, ce qui fait perdre un grand avantage de la programmation objet.

- A l'inverse des variables statiques, l'utilisation du constructeur ***Thread(Runnable)*** offre l'avantage de ne pas partager les informations entre toutes les instances de la classe mais seulement entre les threads issus d'un même objet (qui doit implémenter **Runnable**). Les inconvénients étant que toutes les variables membres sont partagées et qu'il est nécessaire de créer deux objets au lieu d'un pour créer un thread

Chapitre 3- Les Threads en Java

Plan

Synchronisation

- Les Threads partagent les données entre eux. Le programmeur doit être vigilant dans l'utilisation de ces objets partagés, d'où la nécessité de la synchronisation des différents Threads.

- Java permet de verrouiller un objet pour empêcher les accès concurrents.

- La synchronisation d'un bloc d'instruction est la pose d'un verrou sur un objet. La portée de ce verrou est définie par le mot clé **synchronized**, et il n'est levé qu'après la dernière instruction. Les autres instances du thread ne pourront pas effectuer d'action sur cet objet et attendront que le verrou soit relâché pour poursuivre leur traitement.

Synchronisation

Synchronisation d'une partie du code

- Java permet de verrouiller un objet (pas une variable de type primitif) pour empêcher les accès concurrents. Lorsqu'une méthode d'instance qualifiée de **synchronized** est invoquée sur un objet par un thread, elle pose un verrou sur l'objet. Ainsi, si un autre thread invoque une méthode synchronized, elle devra attendre que le verrou soit relâché. Le verrou est relâché si :
  - le code synchronisé a fini de s'effectuer,
  - la méthode wait est invoquée, que nous verrons plus loin.

Synchronisation

Exemple Le programme suivant décrit l'utilisation d'un même mégaphone par 3 orateurs. Chaque Orateur attendra la fin de l'utilisation du mégaphone par le précédent. L'objet de la classe Megaphone est verrouillé dès qu'il est pris par un Thread Orateur, même si celui-ci n'utilise pas le mégaphone à plein temps.

```
Class Megaphone {/*si un orateur utilise le mégaphone alors ce dernier n'est pas disponible pour un autre orateur /
    synchronized void parler(String qui, String quoi, Thread t){
        for (int i=0; i< 10; i++)
        { System.out.println(qui + «  affirme : » + quoi + «  » + i);
          try{ t.sleep(100);}
           catch(InterruptedException e) { System.err.println(e); }}}
Class  Orateur extends Thread {
    String nom, discours; Megaphone m;
    public Orateur(String s, String d, Megaphone m){
        nom =s; discours = d; this.m = m;}
    public void run() {
        m.parler(nom, discours, this);}}
Class  Reunion {
        public static void main(String args[]) {
                Megaphone m = new Megaphone();
                Orateur o1 = new Orateur( "Orateur 1", " premier" , m);
                Orateur o2= new Orateur(" Orateur 2", " deuxième" , m);
                Orateur o3 = new Orateur("Orateur 3", "troisième" , m);
                o1.start(); o2.start(); o3.start(); } }
```

```
Execution
Orat premier 0
….
Orat premier 9
Orat deuxi 0
…
Orat deux 9
Orat trois 0
…
Orat troisi 9
Sans sanchronized
on aura:
Orat premier 0
Orat deux 0
Orat troisi 0
```

Synchronisation

Synchronisation d'une partie du code

Exemple

```
Class Synch implements Runnable {
        public void run() {
                for(int  i = 0; i < 4; i++) {
                        move();
                        }
                }
        synchronized move() {
                System.out.println(Thread.currentThread().getName() + " entre");
                System.out.println(Thread.currentThread().getName() + " sort");
                }
 main(String[] args) {
         Synch o = Synch();
        Thread(o).start();
        Thread(o).start(); } }
```

Exécution

```
Thread-0 entre
Thread-0 sort
Thread-1 entre
Thread-1 sort
Thread-0 entre
Thread-0 sort
Thread-1 entre
Thread-1 sort
```

On observe qu'il n'y a qu'un seul thread à la fois qui entre dans la méthode, l'autre thread attendant que le verrou sur la méthode soit libéré quand l'autre thread sort de celle-ci

Si vous retirez le mot clé **synchronized**, il est fortement probable de voir les deux threads entrer dans la méthode ***move()***, sans qu'un des deux n'en soit encore sorti.

Synchronisation

Interruption avec synchronized

- Lorsque la méthode wait() est invoquée à partir d'une méthode synchronized, en même temps que l'exécution est suspendue, le verrou posé sur l'objet par lequel la méthode a été invoquée est relâché. Dès que le réveil survient, le thread attends de pouvoir reprendre le verrou et continuer l'exécution
- Dans l'exemple qui suit, les orateurs peuvent interrompre leur discours de temps en temps et libèrer le mégaphone pour les orateurs en attente

```
Class Megaphone {
        synchronized void parler(String qui, String quoi, Thread t){
                System.out.println(« mégaphone demandé par «  + qui);

                for (int i=0; i< 10; i++)
                { System.out.println(qui + «  affirme : » + quoi );
                  notifyAll();

                   try{ wait();}
                   catch(InterruptedException e) { System.err.println(e); }}
        }}
```