

# Programmation concurrente en Java

## Thread (fil)

- **thread:**  
terminologie adoptée dans le contexte des systèmes d'exploitation;  
correspond à la notion de processus (Portal, Modula-2) ou de tâche (Ada).
- **création d'un thread:**
  - déclarer une classe *C* comme sous-classe de la classe *Thread* ;
  - dans la classe *C*, redéfinir (*override*) la méthode *run* ;
  - créer une instance *i* de la classe *C* et exécuter la méthode *i.start()* .
- **terminaison d'un programme:**  
liée à l'attribut *user* ou *daemon* des threads. Un programme Java se termine lorsque tous les threads *user* sont terminés (les threads *daemon* n'ont pas besoin d'être terminés).

## Plan

Ce chapitre n'est pas un cours sur Java!

Uniquement les aspects liés à la concurrence:

- threads;
- synchronisation;
- allocation du processeur (scheduling).

Bibliographie:

- S. Oaks, H. Wong, *Java Threads*, O'Reilly, 1997
- D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 1997

## Interface Runnable

```
public abstract interface Runnable {
    // public instance methods
    public abstract void run();
}

class ThreadTest implements Runnable {
    // Runnable définit une méthode run()
    // qui a la même fonction que
    // Thread.run()
    // on y met les instructions qui doivent
    // être exécutées avant le lancement du
    // thread

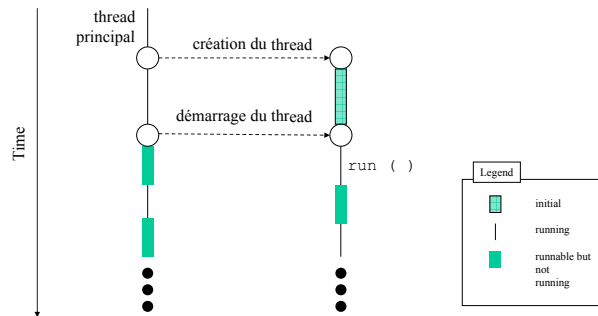
    public void run() {
        // insérer les instructions nécessaires ici
    }
}

Thread myThread;
ThreadTest test;

test = new ThreadTest;
// crée un nouvel objet de type ThreadTest

myThread = new Thread(test);
// crée un nouveau thread dont la méthode
// run invoque test.run

myThread.start();
// lance le thread
```



Programme principal contenant la création d'un thread producteur:

```

class test {
    ...
    public static void main (...) {
        ...
        Producteur p = new Producteur (...);
        //création du thread
        p.start () ; // démarrage du thread qui
                    // execute la procedure run ()
    }
}

```

## Exemple

Déclaration d'une classe producteur:

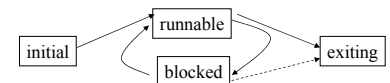
```

class Producteur extends Thread {
    ...
    public void run () {
        ...
        for ( ;; ) {
            // procedure data
            tampon.déposer (...);
            sleep (...); // méthode de la classe
                        // Thread
        }
    }
}

```

## Etats d'un thread

- **initial** : depuis sa création jusqu'à l'appel de la méthode `start ()` de l'objet thread;
- **runnable** : une fois que la méthode `start ()` a été appelée (c'est l'état par défaut);
- **blocked** : état d'un thread qui ne peut s'exécuter pour une raison logique (attente d'E/S, synchronisation, etc.);
- **exiting** : à la fin de la méthode `run ()`.



## Threads de la JVM

En plus des threads du programme Java, une JVM contient les threads suivants:

- **gestionnaire d'horloge**: gère tous les événements internes liés au temps;
- **thread oisif** (priorité 0): exécuté uniquement si aucun autre thread ne s'exécute;
- **ramasseur de miettes (garbage collector)** (priorité 1): récupère les objets qui ne sont plus accessibles (référéncés). Ce thread ne s'exécute que lorsqu'aucun autre thread (à part le thread oisif) ne s'exécute;
- **thread de finalisation**: appelle la méthode `finalize ( )` des objets récupérés par le ramasseur de miettes.

## Synchronisation

Proche du mécanisme des moniteurs, quoique plus primitif (pas de signaux).

L'exclusion mutuelle s'obtient:

- soit en déclarant une méthode `synchronized` :  

```
public synchronized void method(...) {...}
// peut être une méthode statique
```
- soit en déclarant un bloc `synchronized` :  

```
synchronized (O) {...}
// ou synchronized (this) {...}
// (verrou sur l'instance O d'une classe C)
synchronized (C.class) {...}
// (verrou « sur la classe »)
```

## Méthodes de contrôle des threads

### Méthodes statiques:

agissent sur le thread appelant

- `Thread.sleep(long ms)`  
bloque le thread appelant pour la durée spécifiée;
- `Thread.yield()`  
thread appelant relâche le CPU au profit d'un thread de même priorité;
- `Thread.currentThread()`  
retourne une référence sur le thread appelant;

### Méthodes d'instance:

Producteur `p = new Producteur (..)`;

- `p.start()`  
démarré le thread `p`;
- `p.isAlive()`  
détermine si `p` est vivant (ou terminé);
- `p.join()`  
bloque l'appelant jusqu'à ce que `p` soit terminé;
- `p.setDaemon()`  
attache l'attribut « daemon » à `p`;
- `p.setPriority(int pr)`  
assigne une priorité à `p`;
- `p.getPriority()`  
retourne la priorité de `p`;
- ... (beaucoup d'autres)

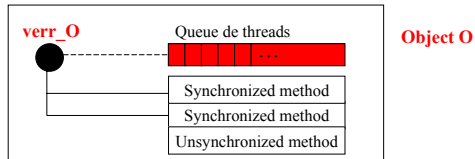
## Exemple de bloc *synchronized*

```
System.out.print (new Date ( ) ) ;
System.out.print ( " : " ) ;
System.out.println (...) ;
```

```
Synchronized (System.out) {
    System.out.print (new Date ( ) ) ;
    System.out.print ( " : " ) ;
    System.out.println (...) ;
}
```

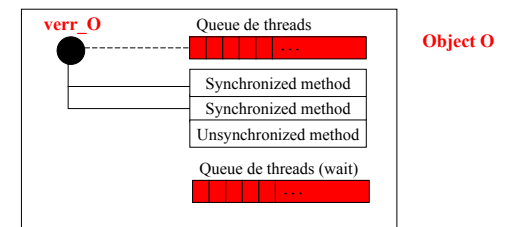
## Explications:

- un verrou **verr\_O** est associé à chaque objet **O** ;
- lorsqu'une méthode **synchronized** de **O** est appelée, le thread doit obtenir le verrou **verr\_O** ;
- une méthode « normale » de **O** n'est pas concernée .



- `wait()` bloque le thread et rend simultanément le verrou `verr_O` ;
- `wait(long timeout)` comme `wait()` , mais si le signal n'a pas été envoyé après `timeout` msec le processus est débloquent ;
- `notify()`
  - débloquent un des threads bloqués par `wait()` (sélection aléatoire), tout en conservant le verrou `verr_O` ;
  - le thread débloquent doit réobtenir le verrou `verr_O` ;
  - `notify()` n'a aucun effet si aucun thread n'attend le signal ;
- `notifyAll()` débloquent tous les threads bloqués par `wait()` (mêmes remarques que pour `notify`)

- La synchronisation est exprimée grâce aux méthodes suivantes de la classe **Object** :
  - `wait()`
  - `wait(long timeout)`
  - `notify()`
  - `notifyAll()`
- `wait()`, `notify()` et `notifyAll()` ne peuvent être appelées que dans une méthode `synchronized`.



## Exemple

---

```
class tampon {
...
    Object tamp [ ] ;
    int inpt, outpt, nbmsg, tailleTamp;
...
    public synchronized void déposer (Object m) {
        while (nbmsg == tailleTamp) { wait() ; }
        tamp[inpt] = m;
        inpt = (inpt + 1) % tailleTamp;
        nbmsg = nbmsg + 1;
        notifyAll() ;
    } // déposer
}
```

---

## Appels imbriqués

---

```
class A {
    public synchronized void m1 ( ) {
        ...
        m2 ( ) ;
        ...
    }
    public synchronized void m2 ( ) {
        ...
        wait ( ) ;
        ...
    }
}
```

---

Aucun problème!

---

```
public synchronized Object prélever ( ) {
    Object m;

    while (nbmsg == 0) { wait() ; }
    m = tamp[outpt] ;
    outpt = (outpt+1) % tailleTamp;
    nbmsg = nbmsg - 1;
    notifyAll() ;
    return m;
} // prélever

} // tampon
```

---



---

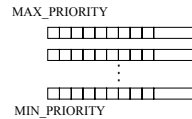
Attention à l'appel imbriqué de méthodes synchronisées:

- un thread  $T_1$  appelle la méthode synchronisée **O1.m1** d'un objet **O1**;
- depuis **O1.m1**,  $T_1$  appelle la méthode synchronisée **O2.m2** d'un objet **O2**, et se met en attente (**wait()**):
  - le verrou **verr\_O2** est relâché ;
  - le verrou **verr\_O1** n'est par contre pas relâché!!!

⇒ risque d'interblocage

## Scheduling: la théorie

- les threads **runnable** se trouvent dans les queues de différentes priorités, gérées par le run-time de Java:
  - si plus d'un thread *runnable*: Java choisit le thread de plus haute priorité;
  - si plus d'un thread de plus haute priorité, Java choisit un thread arbitrairement;
  - un thread *running* perd le CPU au profit d'un thread de plus haute priorité qui devient *runnable*;



- par défaut, un thread a la même priorité que le thread qui l'a créé;
- la priorité peut être changée en appelant **thread.setPriority()**.

## Scheduling: la réalité

- la politique d'allocation du processeur aux threads de même priorité n'est pas fixée par le langage.  
Elle peut être *avec* ou *sans* préemption!!!
- pas de traitement cohérent de la priorité:  
exemple: la priorité n'est pas utilisée par la méthode **notify()** pour choisir le processus à débloquent;
- la spécification du langage n'exige pas que les priorités soient prises en compte par le runtime.