

Thread et synchronisation en Java

Jérôme David - Florian Rodriguez

Introduction

- Thread : « processus léger »
 - Un processus peut contenir plusieurs Thread
 - Tous les threads d'un processus partagent son espace mémoire
 - Chaque thread a son propre pointeur d'exécution (Program Counter)
- Permet de paralléliser des traitements
 - Par défaut, il y a au moins un thread pour l'exécution de l'application, un ou plusieurs pour le ramasse-miettes
 - Inconvénients : utilise des ressources

Contenu d'un Thread

- Du code, un algo à exécuter
 - Dans la méthode `public void run()`
 - Méthode à redéfinir
- Une méthode pour le démarrer
 - `public void start()`
- Un thread vit jusqu'a la fin de sa méthode `run()`
 - Endormir un thread : `void sleep(int milisec)`
 - Mettre en attente un thread : `void wait()`

Création des threads

- Pour créer un bon vieux thread, il existe deux façons :
 - Dériver la classe `Thread` (et redéfinir la méthode `run()`)
 - Implémenter l'interface `Runnable` (et définir la méthode `run()`)

Dérivation de la classe Thread ou implémentation de Runnable

Dérivation de la classe Thread

```
class MonThread extends Thread {  
  
    public MonThread() {  
        // initialisation  
    }  
  
    public void run() {  
        // le code de l'action du thread  
    }  
}
```

Copy

```
MonThread t = new MonThread();  
t.start(); // execute la methode run() de MonThread en  
           // parallele au thread de ce programme
```

Copy

Implémentation de l'interface Runnable

```
class MaTache implements Runnable {  
  
    public MaTache() {  
        // initialisation  
    }  
  
    public void run() {  
        // le code de l'action du thread  
    }  
}
```

Copy

```
MaTache tache = new MaTache();  
Thread t = new Thread(tache) ;  
t.start(); // execute la methode run() de MaTache en  
           // parallele au thread de ce programme
```

Copy

La classe Thread - méthodes statiques -

- Méthodes qui agissent sur le thread qui exécute la méthode
 - `Thread.sleep(long milisec)` : endort le thread courant
 - `Thread.currentThread()` : retourne une référence vers le thread courant
 - `Thread.yield()` : le thread passe son tour

La classe Thread - méthodes d'instances -

```
Thread t = new Thread();
```

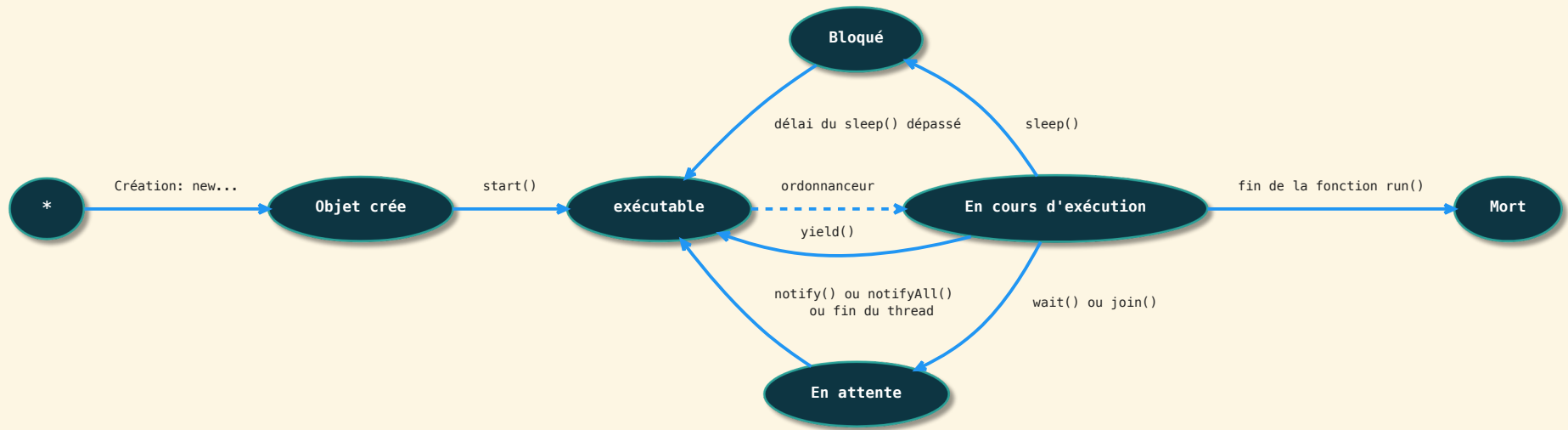
[Copy](#)

- `t.start()` : démarre le thread
 - Méthode non bloquante (redonne tout de suite la main)
 - Exécute le contenu de `run()` en parallèle
- `t.join()` : bloque le thread appelant tant que `t` n'est pas terminé
- `t.yield()` : le thread `t` passe son tour pour être exécuté
- Etc...

La synchronisation

- Se fait par le moniteur d'un objet
 - Moniteur = verrou + liste d'attente
 - Tout objet possède un moniteur
- Etant donné un `Object o`
 - `o.wait()` : bloque thread qui l'exécute
 - Le thread est mis dans la file d'attente du moniteur de `o`
 - `o.notify()` : réveille un des threads qui attend
 - `o.notifyAll()` : réveille tous les threads qui attendent
 - Ces méthodes peuvent lever des `InterruptedException`

Cycle de vie d'un Thread



Interruption d'une tâche

- Un thread bloqué peut être interrompu via la méthode `void interrupt()`

```
class MonThread extends Thread {  
    public void run() {  
        try {  
            Thread.sleep(100);  
            System.out.println("Je me suis réveillé tout seul");  
        }  
        catch (InterruptedException e) {  
            System.out.println("On m'a réveillé :-(");  
        }  
    }  
}
```

Copy

```
MonThread t = new MonThread();  
t.start();  
Thread.sleep((long) (Math.random()*200));  
t.interrupt();
```

Copy

L'exclusion mutuelle

- Mot-clé synchronized

```
public class ExclusionMutuelle {  
    public static void main(String[] args) throws InterruptedException {  
        Compte c = new Compte();  
        Thread[] lesThreads = new Thread[10];  
        for (int i=0 ; i<lesThreads.length ; i++) {  
            lesThreads[i] = new Thread() {  
                public void run() {  
                    for (int i=0 ; i<1000 ; i++) {  
                        c.deposerUnEuros();  
                    }  
                }  
            };  
            lesThreads[i].start();  
        }  
        for (Thread x : lesThreads) {  
            x.join();  
        }  
        System.out.println(c.value);  
    }  
}
```

Copy

```
class Compte {  
    public volatile int value;  
  
    // methode appelée en exclusion mutuelle  
    // grâce au mot clé synchronized  
    public synchronized void deposterUnEuros() {  
        value+=1;  
    }  
}
```

Copy

Synchronized

- Utilisations du mot clé synchronized
 - Devant une méthode (verrouille l'objet sur lequel est appelée la méthode) :

```
class Compte {  
    public volatile int value;  
    public synchronized void deposerUnEuros() { value+=1; }  
}
```

Copy

- En tant que bloc :

```
class Compte {  
    public volatile int value;  
    public Object lock = new Object();  
    public void deposerUnEuros() {  
        synchronized(lock) { value+=1;}  
    }  
}
```

Copy

Interbloquage

- Attention à l'interbloquage si on imbrique des blocs synchronisés

```
class MonThread1 extends Thread {  
    public void run(){  
        synchronized(a){  
            . . .  
            synchronized(b){  
                . . .  
            }  
        }  
    }  
}
```

Copy

```
class MonThread2 extends Thread {  
    public void run(){  
        synchronized(b){  
            . . .  
            synchronized(a){  
                . . .  
            }  
        }  
    }  
}
```

Copy

Attente passive

- La méthode `wait()` appelée sur un objet `o`, met le thread appelant en attente
 - `o.wait()`, `o.wait(long millisec)`, ...
 - Doit être appelée en exclusion mutuelle avec `synchronized (o) { o.wait() ; }` par exemple
 - Peut lever une `InterruptedException`
- Pour « réveiller » le thread en attente, un autre thread doit appeler
 - `o.notify()` : réveille un des threads en attente (on ne contrôle pas lequel)
 - `o.notifyAll()` : réveille tous les threads en attente

Exemple

- Démo

Les sémaphores

- Exercice : implémenter une classe sémaphore avec les moniteurs
- Il existe une classe toute prête pour cela : `java.util.concurrent.Semaphore`
- `Semaphore sem = new Semaphore(int nbJetons);`
 - `sem.acquire();`
 - pour prendre un jeton, avec attente si non disponible
 - `boolean res = sem.tryAcquire();`
 - prendre un jeton si disponible sans attendre
 - On peut passer aussi un délai d'attente en paramètre
 - `sem.release();`
 - pour relâcher le jetons

Limites du synchronized

- Porte sur une méthode maximum
 - On ne peut pas verrouiller dans une méthode et déverrouiller dans une autre
- Le relâchement de verrous se fait dans l'ordre opposé
 - premier verrou obtenu, dernier relâché
- Une seule file d'attente sur un moniteur
- Pas de gestion de l'équité
 - le thread qui attend depuis le plus longtemps n'est pas nécessairement servi en premier

ReentrantLock

- Verrou avec extensions par rapport au moniteurs + synchronized

```
class X {  
    private final ReentrantLock lock = new ReentrantLock();  
  
    public void m() {  
        lock.lock(); // prise de verrou bloquante  
        try {  
            // ... code en exclusion mutuelle  
        } finally {  
            lock.unlock(); // relâche le verrou  
        }  
    }  
}
```

Copy

```
class Y {  
    private final Object lock = new Object();  
    // ...  
  
    public void m() {  
        synchronized(lock) { // prise de verrou bloquante  
            // ... code en exclusion mutuelle  
        } // relache le verrou  
    }  
}
```

Copy

Synchronisation avec ReentrantLock

- On peut obtenir plusieurs conditions (files d'attente) sur un verrou

```
class Z {  
    private final ReentrantLock lock = new ReentrantLock();  
    private final Condition cond = lock.newCondition();  
  
    public void arriveeRDV() throws InterruptedException {  
        lock.lock();  
        try {  
            cond.await(); //attendre  
        } finally {  
            lock.unlock();  
        }  
    }  
  
    public void debloquerRDV() throws InterruptedException {  
        lock.lock();  
        try {  
            // reveiller  
            cond.signal(); //cond.signalAll();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Copy

Autres classes pour synchronisation

- Les barrières : bloque des threads en un point jusqu'à ce qu'a ce que le nombre requis de threads aient atteint ce point
 - `CyclicBarrier` , `CountDownLatch`