



Thread Java et la synchronisation

Les processus légers en Java

- Un programme → Un processus
 - Exécution et lecture des instructions de manière séquentielle
- Java permet l'exécution concurrente de threads
 - Un processus → plusieurs sous-processus
 - Sous-processus : processus légers ou fils d'exécution (threads)
 - espace d'adressage commun
- Les programmes qui utilisent plusieurs threads sont dits multithreadés

Processus légers

Création - Lancement



1. Dériver la classe *Thread* (*java.lang*)

- *Redéfinir la méthode `run()` qui contient les instructions qu'exécutera le thread*

→ Instancier la classe définie

→ Lancer le thread via la méthode *start()* qui invoquera la méthode *run()* redéfinie

Processus légers

Création - Lancement

2. Implémenter l'interface *Runnable* (*java.lang*)

- Définir la méthode *run()* qui contient les instructions qu'exécutera le thread

- Instancier la classe définie
- Créer une instance de la classe *Thread* en passant cet objet en paramètre
- Lancer le thread via la méthode *start()*

3. Implémenter l'interface *Callable* (*java.util.concurrent*) [java 1.5]

- Interface similaire à *Runnable*
- renvoie un résultat sinon lève une exception

```
public class FactorielleThread extends Thread {
    private int nb;

    public FactorielleThread(int n) { this.nb = n;}

    public void run(){
        int res = 1;
        for (int i=1; i<=nb; i++)
            res = res * i;
        System.out.println("Factorielle de " + nb + " = " +res)
    }
}
```

Test

```
FactorielleThread factT ;
for (int i=10; i >= 1; i--) {
    factT = new FactorielleThread(i);
    factT.start();
}
```

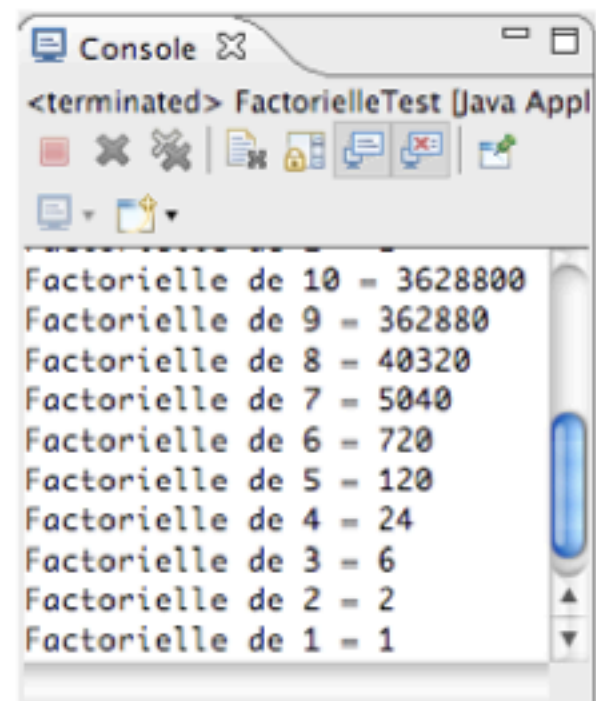
```
public class FactorielleRunnable implements Runnable{
    private int nb;

    public FactorielleRunnable(int n) { this.nb = n;}

    public void run(){
        int res = 1;
        for (int i=1; i<=nb; i++)
            res = res * i;
        System.out.println("Factorielle de " + nb + " = " +res);
    }
}
```

Test

```
FactorielleRunnable fc ;
for (int i=10; i >= 1; i--) {
    fc = new FactorielleRunnable(i);
    Thread factR = new Thread (fc);
    factR.start();
}
```



```
Console X
<terminated> FactorielleTest [Java Appli]
Factorielle de 10 = 3628800
Factorielle de 9 = 362880
Factorielle de 8 = 40320
Factorielle de 7 = 5040
Factorielle de 6 = 720
Factorielle de 5 = 120
Factorielle de 4 = 24
Factorielle de 3 = 6
Factorielle de 2 = 2
Factorielle de 1 = 1
```

Constructeurs de Thread

Constructors

Constructor and Description

`Thread()`

Allocates a new Thread object.

`Thread(Runnable target)`

Allocates a new Thread object.

`Thread(Runnable target, String name)`

Allocates a new Thread object.

`Thread(String name)`

Allocates a new Thread object.

`Thread(ThreadGroup group, Runnable target)`

Allocates a new Thread object.

`Thread(ThreadGroup group, Runnable target, String name)`

Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

`Thread(ThreadGroup group, Runnable target, String name, long stackSize)`

Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size.

`Thread(ThreadGroup group, String name)`

Allocates a new Thread object.

Quelques méthodes

Méthodes	Description
<code>run()</code>	Méthode contenant l'ensemble des tâches à exécuter
<code>start()</code>	Démarre la thread et invoque la méthode <code>run()</code>
<code>currentThread()</code>	Retourne une référence au thread actuellement traité par le processeur
<code>destroy()</code>	Arrête définitivement le thread
<code>sleep(long)</code>	Met le thread en attente durant le temps exprimé en ms fourni en paramètre
<code>wait()</code> , <code>wait(long)</code>	Méthodes héritées de la classe <i>Object</i> qui obligent tous les threads autres que le thread courant tc à attendre que tc ait terminé son exécution ou qu'une invocation de la méthode <code>notify()</code> soit effectuée
<code>yield()</code>	Indique que le thread peut être suspendu pour permettre l'exécution des autres threads
<code>notify()</code> , <code>notifyAll(long)</code>	Méthodes héritées de la classe <i>Object</i> qui indique à un autre/ tous les thread(s) que la voie est libre
<code>join()</code> , <code>join(long)</code>	Attend (au plus le nombre de ms fourni en paramètre) la fin de l'exécution du thread

États d'un thread

- Un thread est considéré comme un objet
 - comportement (spécifié par ses méthodes) lui permettant de passer d'un état à un autre
- États d'un thread
 - créé
 - Exécutable
 - Prêt à être exécuté
 - en cours d'exécution
 - Bloqué
 - En attente
 - Temporisé
 - Terminé

La méthode getState() renvoie

NEW

RUNNABLE

BLOCKED

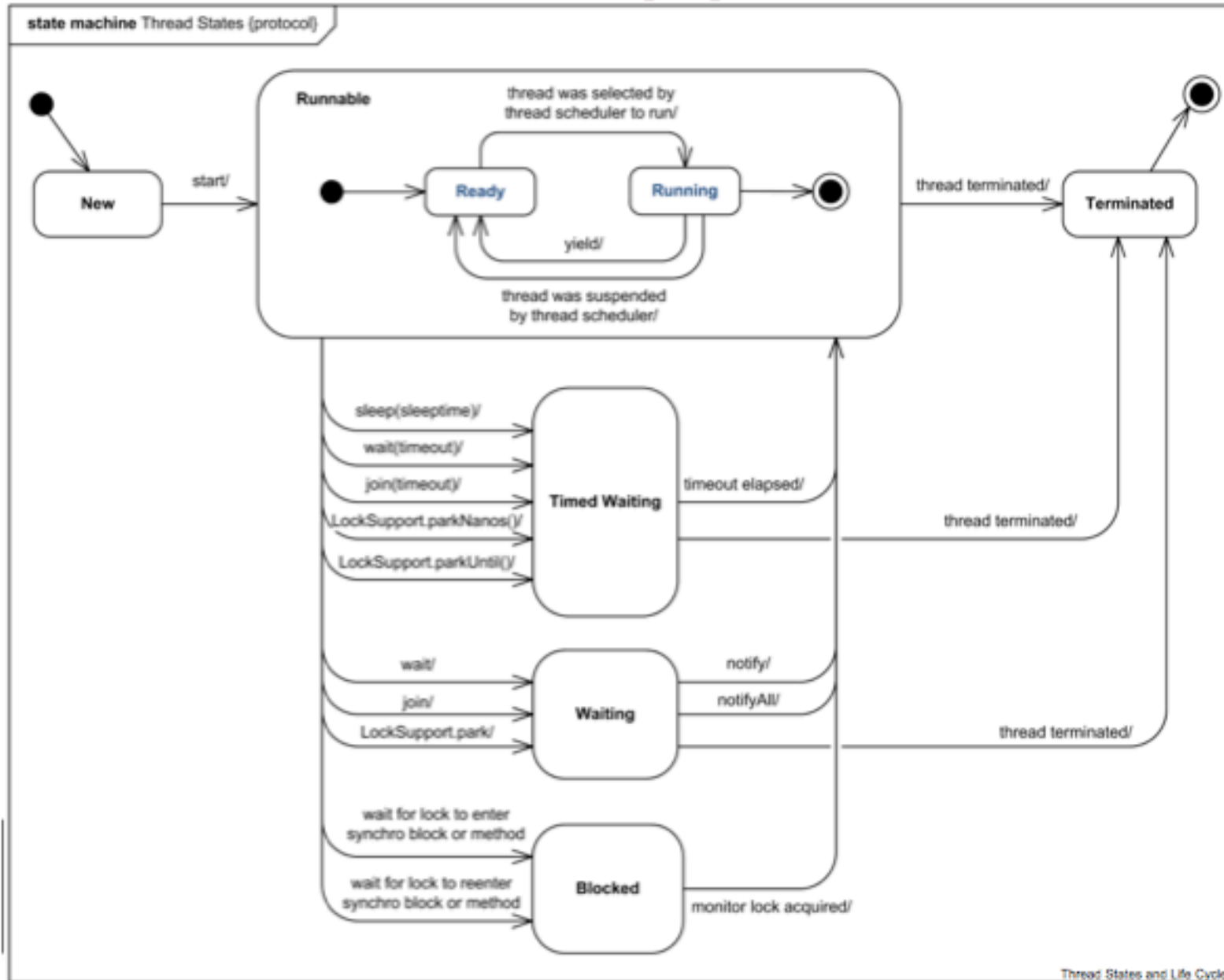
WAITING

TIME_WAITING

TERMINATED

NB! À un instant donné, un thread ne peut être que dans UN seul de ces six états

États d'un thread (2)



Thread States and Life Cycle in Java 6 : <http://www.uml-diagrams.org/>

Les processus légers en Java

■ Un thread passe la main :

- sur une E/S
- sur un objet synchronisé
- pour céder la main explicitement → `yield()`
- pour faire une pause pendant une certaine durée → `sleep(1000)`
- pour attendre un autre thread → `join()`
- lorsqu'il attend notification → `wait()`
- lorsqu'il se termine

Les priorités

- Lors de la création d'un thread
 - sa priorité est égale à celle du thread dans lequel il est créé
 - On lui attribue la priorité moyenne si il n'est pas créé dans un autre thread
 - On peut lui affecter une autre priorité
 - *setPriority(int)*
- La priorité d'un thread varie entre 1 et 10
 - Priorité par défaut : 5 (NORM_PRIORITY)
 - Trois constantes : MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY



Les threads peuvent avoir des priorités différentes

Ordonnancement des threads

- Exécution d'une machine virtuelle Java via un processus du système d'exploitation
 - Ce processus exécute plusieurs threads Java
 - Le thread principal
 - `static void main(String argv[])`
 - Les threads créés par le programme
 - Garbage collector
 - ...



L'ordonnancement des threads Java
n'est pas précisément spécifié

Ordonnancement des threads

- L'ordonnancement dépend du système d'exploitation
 - Java est un langage multi-plateformes (Windows, Linux, Mac OS, Solaris, ...)
 - Systèmes d'ordonnancement possibles : préemptif, coopératif, round-robin (tourniquet)
- En Java
 - un thread plus prioritaire à la main en priorité (préemptif)
 - Les threads de même priorité

 ***Ne pas se reposer sur un système d'ordonnancement particulier pour le développement d'applications multithreadées***

Interactions entre threads

■ Les threads sont des objets

- Ils possèdent des références sur d'autres objets
- Un thread peut appeler des méthodes sur ces objets
- Possibilité de faire appel à des méthodes sur le thread
- Communication/interaction possible via ces objets ou les méthodes du thread

Partage de données

- Les threads s'exécutent sur la même machine virtuelle, dans le même espace mémoire
- Accès possible aux mêmes objets

Interactions entre threads

```
public class Synchro implements Runnable {
    private int j = 1;
    public void run() {
        for (int i = 1; i <= 30; i++) {
            System.out.println(Thread.currentThread().getName() + " " + j);
            j++;
        }
    }

    public static void main (String[] args) {
        Synchro s = new Synchro();
        Thread a = new Thread(s);
        Thread b = new Thread(s);
        Thread c = new Thread(s);
        a.start();
        b.start();
        c.start();
    }
}
```

traces d'exécution
possibles

Thread-1 57
Thread-1 58
Thread-1 59
Thread-1 60
Thread-2 61
Thread-2 62
Thread-2 63
Thread-2 64

Thread-1 57
Thread-1 58
Thread-1 59
Thread-1 60
Thread-2 60
Thread-1 62
Thread-2 63
Thread-2 64

Thread-1 33
Thread-1 34
Thread-1 35
Thread-2 36
Thread-2 37
...
Thread-2 64
Thread-2 65
Thread-1 36
Thread-1 67
Thread-1 68

Interactions entre threads

■ Cas possibles :

```
Thread-0 : System.out.println(j);  
Thread-0 : j++;  
Thread-1 : System.out.println(j);  
Thread-1 : j++;
```

```
Thread-0 : System.out.println(j);  
Thread-1 : System.out.println(j);  
Thread-0 : j++;  
Thread-1 : j++;
```

```
Thread-0 : System.out.println(j);  
Thread-1 : System.out.println(j);  
Thread-0 : j++;  
Thread-0 : System.out.println(j);  
Thread-1 : j++;  
Thread-1 : System.out.println(j);  
Thread-0 : j++;
```

Problème d'accès concurrents à
des données partagées

→ solution : la synchronisation

Sections critiques



Une section critique est une zone de programme où se produit un accès concurrent et dont le résultat des traitements et fonction de l'ordonnancement des tâches ou des processus

Exclusion mutuelle



Plusieurs sections critiques dépendantes ne doivent jamais exécuter leur code simultanément (par plusieurs threads différents) : on dit qu'elles sont en exclusion mutuelle.

Synchronisation en Java



- Mécanismes pour protéger une section critique d'un accès simultané
 - ▶ Utilisation du mot clé synchronized
 - ▶ Utilisation de la classe ReentrantLock

Le mot clé Synchronized

- La méthode la plus simple
- Elle s'applique sur un objet (n'importe lequel)
- Exclusion mutuelle sur une séquence de code
 - ▶ Il est impossible que 2 threads exécutent en même temps une section de code marquée synchronized pour un même objet
 - ▶ Sauf si un thread demande explicitement à se bloquer avec un wait()

```
public synchronized void uneMéthode() {  
    // section principale  
}
```


Le mot clé Synchronized

- Verrou intrinsèque
 - ▶ chaque objet possède un verrou intrinsèque et que le verrou a une condition intrinsèque.
 - ▶ Le verrou gère les threads qui tentent d'entrer dans une méthode synchronized.
 - ▶ La condition gère les threads qui ont appelée la méthode wait().

```
public synchronized void uneMéthode() {  
    // section principale  
}
```

Le mot clé Synchronized

- Deux utilisations de synchronized
 - Sur la méthode d'une classe (s'applique à tout son code pour un objet de cette classe)

```
public synchronized void uneMéthode() {  
    // section principale  
}
```

- Sur un objet quelconque

```
public void maMéthode() {  
    .....  
    synchronized(unObjet){  
        // section principale  
    }  
}
```

Limitation du mot clé Synchronized



- 1. Vous ne pouvez pas interrompre un thread qui tente d'acquérir un verrou.*
- 2. Vous ne pouvez pas spécifier de temporisation lorsque vous tentez d'acquérir un verrou.*
- 3. Ne disposez que d'une seule condition par verrou peut se révéler insuffisant.*

La classe ReentrantLock

- Une autre méthode pour réaliser des verrous est apparue dans Java 1.5
- Le paquetage `java.util.concurrent.locks` définit la classe de verrou, `ReentrantLock`.

```
Lock l = new ReentrantLock();  
l.lock();  
try {  
    //section critique  
} finally {  
    l.unlock();  
}
```

ATTENTION!



Le verrou doit être débloqué

- Sinon tous les autres threads (qui ont voulu acquérir le verrou) seront bloqués si la section critique déclenche une exception

La classe ReentrantLock

- Possibilité d'associer des objets de condition aux verrous

- ▶ Utilisation de la méthode newCondition()

```
private Lock verrou = new ReentrantLock();  
private Condition maCondition = verrou.newCondition();
```

```
verrou.lock();  
while(! (ok pour continuer))  
    maCondition.await();  
...  
maCondition.signalAll();  
verrou.unlock();
```

Section critique

- await()
 - désactive le thread qui abandonne le verrou
un autre thread peut alors de le prendre (et influencer sur la condition du while)
 - le thread entre un jeu d'attente pour la condition
- signalAll()
 - réactive tous les threads qui sont en attente de la condition, lesquels vont tenter de réacquérir le verrou