



# Développement Logiciel Java

12/02/2025



# Le multithreading

Certains traitements de notre applications peuvent s'avérer longs et bloquer l'exécution du programme séquentiel.

Ces traitements peuvent être délégués à des flux d'exécutions - les threads - séparés du programme principal et exécutés en parallèle de celui-ci.

Ce parallélisme est réalisé par le système en interrompant régulièrement un flux d'exécution pour donner la priorité à un autre, puis en l'interrompant à son tour, etc...

Cela nécessite de bien gérer les objets partagés entre threads afin de ne pas laisser un autre thread y accéder dans un état inconsistent dû à une interruption.



# Les Threads

Les threads ne sont pas des processus. Ils cohabitent à l'intérieur d'un seul processus et partagent donc l'espace mémoire et les ressources.

Chaque thread dispose de sa propre stack et d'une zone mémoire dédiée (TLS - Thread Local Storage) pour sauvegarder son état en cas d'interruption.



# Créer un Thread

Pour créer un Thread, il faut soit:

- Hériter de la classe Thread
- Implémenter l'interface Runnable

Dans les deux cas, le code d'exécution du Thread sera dans la méthode *public void run()*.



# Créer un Thread

Une fois la classe créée, le thread est démarré de la façon suivante:

```
Thread thread = new Thread(new DemoThread());
thread.start();
```

Depuis n'importe où dans le code, *Thread.currentThread()* renvoie le thread courant.



# Sleep & Yield

La classe statique Thread offre les méthodes:

- *Thread.sleep(time)* : time en ms indique le temps pendant lequel l'exécution du thread doit s'interrompre. Le temps machine est alors alloué à un autre thread.
- *Thread.yield()*: Indique que le thread n'a pas d'activité en cours, il veut bien laisser son temps machine à un autre thread mais récupéré la main dès que possible.

Comme il est possible de demander l'arrêt d'un thread, *Thread.sleep()* est déclaré comme levant l'erreur *InterruptedException*. Cela signifie que notre thread a été stoppé pendant son sommeil, et nous laisse l'opportunité pendant le bloc *catch* de réaliser des opérations de fermeture.



# Join

Lorsqu'un thread démarre un autre thread, il est possible que le premier ait besoin d'attendre la fin de l'exécution du second pour continuer son traitement. Il peut donc appeler `join()` sur la référence du second thread.

L'exécution du premier thread va alors être interrompue en attendant la fin de l'exécution du second.

Comme pour `sleep()`, `join()` peut lever une `InterruptedException`



# Interruption

Il est possible de demander à un thread de s'interrompre grâce à la méthode *interrupt()*.

Le thread secondaire dispose d'une méthode *isInterrupted()*. C'est sa responsabilité de tester cette méthode le plus souvent possible (aux moments clés de son exécution), de nettoyer ses objets et stopper son traitement s'il constate qu'il doit s'interrompre.

Si le thread à interrompre est actuellement bloqué (notamment par *sleep()* ou *join()*), il recevra une Exception.



# Classes anonymes et Lambdas

Runnable étant une interface fonctionnelle, il est possible de créer un thread avec une classe abstraite ou une expression lambda.

C'est un avantage si le code du thread a besoin d'accéder à des variables de la méthode qui l'appelle, ou des membres de sa classe parent.



# Synchronisation

L'exécution des Threads en parallèle peut poser des problèmes si:

- Plusieurs threads souhaitent modifier le même objet, et l'interruption d'un thread risque de laisser l'objet dans un état instable.
- Plusieurs certains blocs de code doivent s'exécuter d'une traite sans être interrompus par un autre thread.

Le système de synchronisation permet de demander à un thread d'attendre la fin du traitement sensible pour commencer le sien.



# Méthodes Synchronized

Lorsque certaines méthodes d'une classe sont déclarées synchronized, il n'est possible d'exécuter l'une de ces méthodes que par un seul thread à la fois.

```
public synchronized void syncMethod() {  
    // Do something  
}
```

Cela peut poser des problèmes de performance - et même remettre en question l'utilité des threads - si la classe contient beaucoup de méthodes synchronized.

Les constructeurs ne peuvent pas être *synchronized*



# Bloc Synchronized

S'il est nécessaire de mettre en place une synchronisation plus fine que sur les méthodes d'une classe, il est possible de créer des blocs *synchronized* à l'aide d'un objet "verrou".

Le verrou peut être l'objet qui ne doit pas être modifié de manière concurrente, ou n'importe quel objet.

Une fois qu'un thread a la main sur un verrou, celui-ci devient inaccessible aux autres.

Une fois un verrou acquis par un thread, il l'est jusqu'au bout du traitement. Si ce traitement appelle des méthodes qui sont elles-mêmes protégées par le même verrou, alors la phase d'acquisition n'est pas relancée (reentrant lock).



# Semaphore et Mutex

La classe Semaphore est un objet thread-safe permettant de maintenir un nombre de droits d'accès fixe.

Lorsqu'un thread souhaite effectuer une opération limitée par un sémaphore, il doit appeler la méthode `.acquire()` sur celui-ci. Si aucun droit n'est disponible, la méthode est bloquée jusqu'à la libération d'un accès.

A la fin de son traitement, le thread appelle `.release()` pour libérer son droit.

Un sémaphore ne supportant qu'un seul droit s'appelle un Mutex.



# Communication entre Threads

La méthode `join()` permet d'attendre la fin de l'exécution d'un thread. Mais des threads peuvent avoir besoin de communiquer pendant leur exécution également.

Pour réaliser ceci, les threads doivent partager un objet. Un thread peut alors se mettre en attente en appelant la méthode `.wait()` sur cet objet. Un autre thread peut réveiller les threads en attente sur l'objet en appelant `.notify()`.

Ces deux méthodes doivent être appelées depuis des méthodes *synchronized*.