# Threads en Java

Karima Ennaoui

karima.ennaoui@insa-cvl.fr

INSA CVL, 4A STI

Bibliographie: Bytecode basics, A first look at the bytecodes of the Java virtual machine, Bill Venners, JavaWorld.com, 09/01/96
https://www.cubrid.org/blog/understanding-jvm-internals/
Inspiré des slides de Jean-François Lalande

Tuesday 11th December, 2018

# Threads

## Définition

Un thread (ou processus léger) est un flot d'exécution partageant l'intégralité de son espace d'adressage avec d'autres processus légers. Il est en général géré à un plus haut niveau, par rapport au système d'exploitation. Les différents processus légers peuvent s'exécuter alors dans un même processus lourd, mais de manière concurrente.
**Exemple:** Les threads de la machine virtuelle Java

# Différence entre un processus et un thread.

| Processus | Thread |
|---|---|
| Plus proches du système d'exploitation. | Exécuté dans l'environnement d'un process. |
| Mémoire dédiée (consommation de ressource). | Espace mémoire partagée entre threads ayant le même parent. |
| Isolation de d'autres processus | Partage inter-thread (Données, Ressources...) |
| Création/Arrêt plus lents | Prend moins du temps |

## Différence entre un processus et un thread.

| Processus | Thread |
|---|---|
| Plus proches du système d'exploitation. | Exécuté dans l'environnement d'un process. |
| Mémoire dédiée (consommation de ressource). | Espace mémoire partagée entre threads ayant le même parent. |
| Isolation de d'autres processus | Partage inter-thread (Données, Ressources...) |
| Création/Arrêt plus lents | Prend moins du temps |

$\Rightarrow$ Le concept de thread a été mis en place afin de surmonter les limites des processus dans **la programmation concurrente**.

## Exemples de threads en Java

- Les threads dans la JVM dont chacun s'occupe de l'execution d'une méthode d'une classe.
- Dans les interfaces graphiques, un listner d'un évenement associé à un objet est un thead qui s'éxecute en background tant que l'interface est active et visible.
  **bouton.addActionListener(new BoutonListener());**

## Création de thread en Java: Inerface Runnable

- Cette interface doit être implémentée par toute classe qui contiendra des traitements à éxécuter dans un thread.
- Une seule méthode : run()
- La classe doit redéfinir cette méthode pour contenir le traitement à exécuter dans le thread → le "main" du thread.

```
public class MyRunnable implements Runnable
public void run() { ... }
```

## Création de thread en Java: Classe Thread

- Définie dans le package java.lang et implémentant l'interface Runnable
- Différents constructeurs prenant un ou plusieurs des paramètres suivants: Nom du thread, un objet Runnable et un groupe auquel sera rattaché le thread.

```
public class MyThread extends Thread {
public void run() { ... }
```

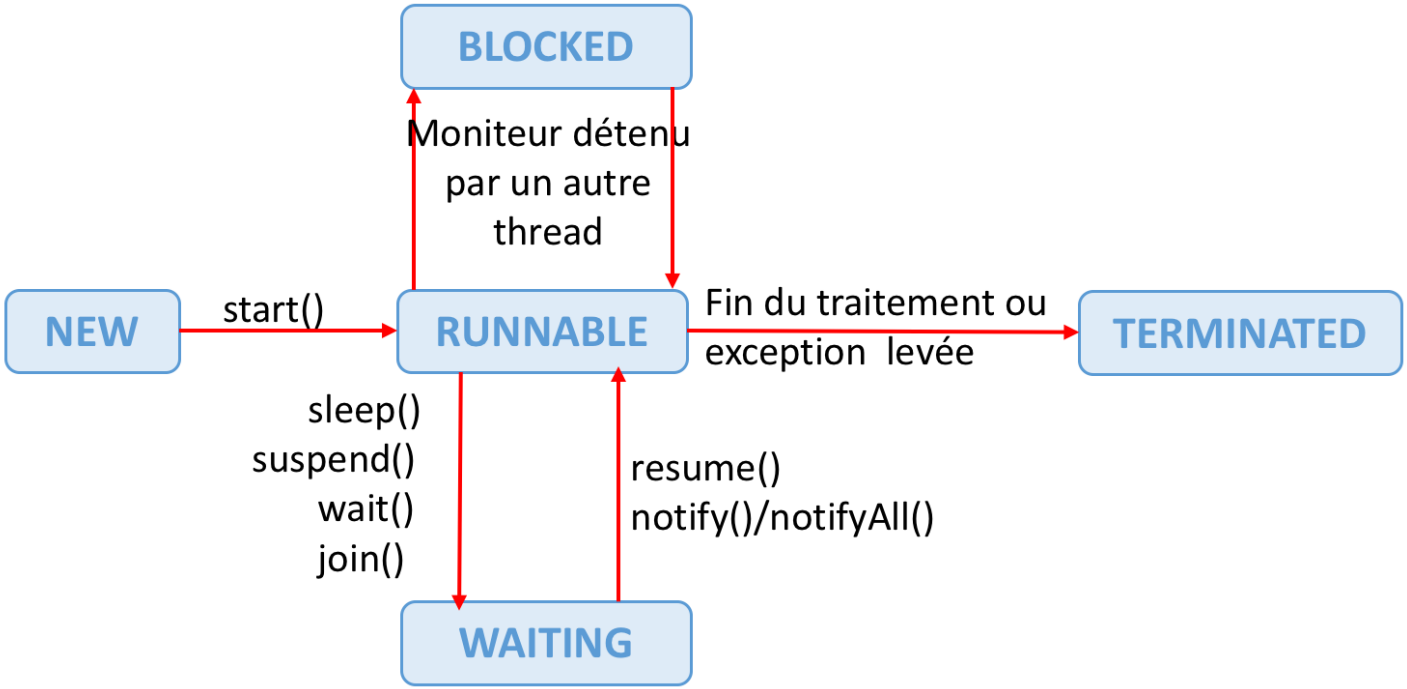
- La classe Thread a deux attributs: le nom et la priorité.

- Instantiation du thread:

```
MyThread t = new MyThread();
t.start();
```

## Cycle de vie d'un thread en Java

- Un objet de type thread (ou extension) suit le cycle de vie suivant:




- le statut est encapsulé dans **Thread.State**

## Changement de statut d'un thread en Java

- A partir de Java 1.5, Les méthodes permettant de stopper ou de suspendre un thread sont dépréciées, pour éviter de laisser des objets en cours de modification dans un état "non-cohérent.
- Seule la possibilité d'interrompre un processus a été gardé: **interrupt()**.
- **interrupt()** peut lever des exceptions si le thread était en pause ou en cours d'utilisation d'entrées/sorties.

## Changement de statut d'un thread en Java

- Un thread peut attendre qu'un autre thread se termine en utilisant la méthode join() pour attendre la terminaison d'un autre fil d'exécution concurrent avant de continuer son propre fil d'exécution.

## Changement de statut d'un thread en Java

- Un thread peut attendre qu'un autre thread se termine en utilisant la méthode join() pour attendre la terminaison d'un autre fil d'exécution concurrent avant de continuer son propre fil d'exécution.
- On peut soi-même demander explicitement l'endormissement d'un thread pendant un temps donné en utilisant **Thread.sleep(int temps)**.

## Changement de statut d'un thread en Java

- Un thread peut attendre qu'un autre thread se termine en utilisant la méthode join() pour attendre la terminaison d'un autre fil d'exécution concurrent avant de continuer son propre fil d'exécution.
- On peut soi-même demander explicitement l'endormissement d'un thread pendant un temps donné en utilisant **Thread.sleep(int temps)**.
- On peut aussi rendre explicitement le processeur pour qu'il soit assigner au prochain thread avec **yield()**.

# Schedulers

- A l'intérieur d'un processus, plusieurs threads peuvent s'exécuter en concurrence $\rightarrow$ Multi-threading.
- Le scheduler s'occupe de l'ordonnancement de la prise du fil d'exécution par chaque thread.
- Dans le cas où plusieurs processeurs sont disponibles, le scheduler s'occupe de répartir n threads sur m processeurs.
- On distingue deux catégories de schedulers:
  - Schedulers coopératifs
  - Schedulers préemptifs

# Schedulers

- A l'intérieur d'un processus, plusieurs threads peuvent s'exécuter en concurrence $\rightarrow$ Multi-threading.
- Le scheduler s'occupe de l'ordonnancement de la prise du fil d'exécution par chaque thread.
- Dans le cas où plusieurs processeurs sont disponibles, le scheduler s'occupe de répartir n threads sur m processeurs.
- On distingue deux catégories de schedulers:
  - Schedulers coopératifs
  - Schedulers préemptifs

**Remarque:** Dans le cas de concurrence inter processus sur un ou plusieurs processeurs, le système d'exploitation donne la main alternativement à chaque processus.

## Schedulers coopératifs

- Les threads s'exécutent jusqu'à ce qu'ils décident de relâcher explicitement le processeur.
- Scheduler passif
- Avantage: Simplicité de la gestion des données partagées.
- Inconvénient: Difficulté de le faire fonctionner sur plusieurs processeurs.

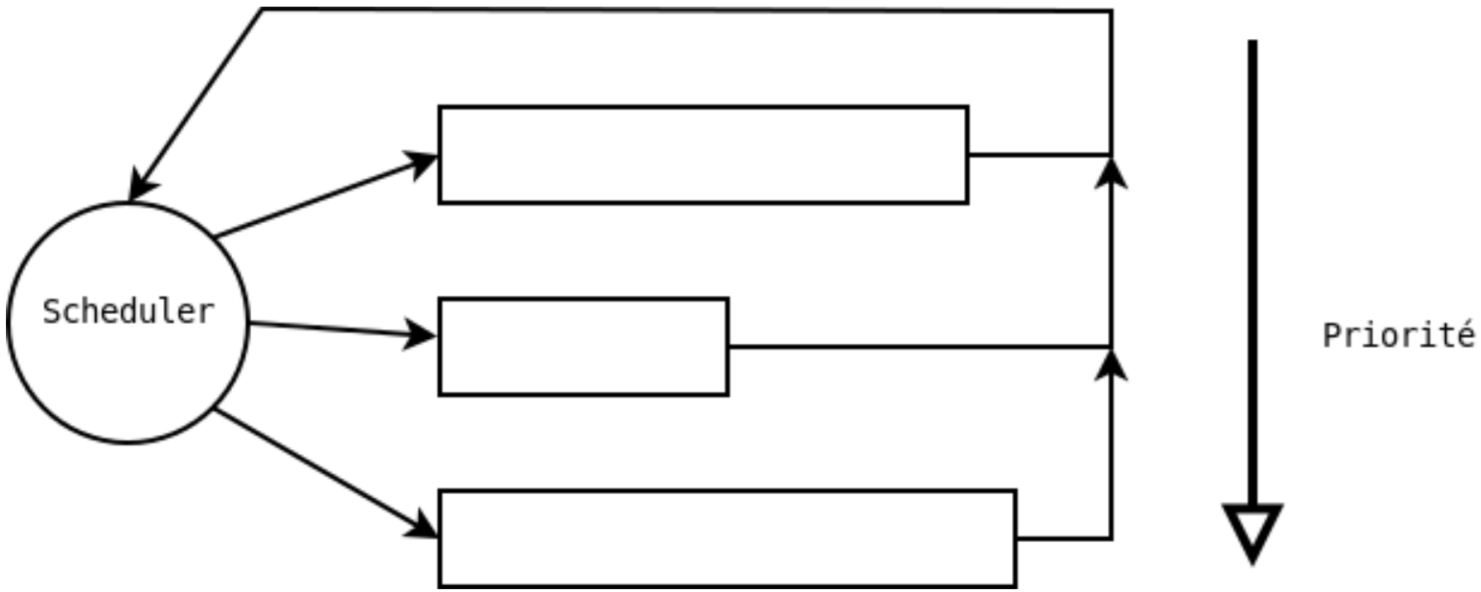
## Schedulers préemptifs

- Un thread particulier ou une partie du système d'exploitation s'occupe de décider quel thread doit s'exécuter.
- Scheduler actif: son rôle est de répartir au mieux les ressources du système (par exemple en fonction des priorités des threads).
- Avantage: gestion de ressource dynamique
- Inconvénient: la difficulté que pose la gestion des données partagées pouvant se trouver dans un état incohérent.

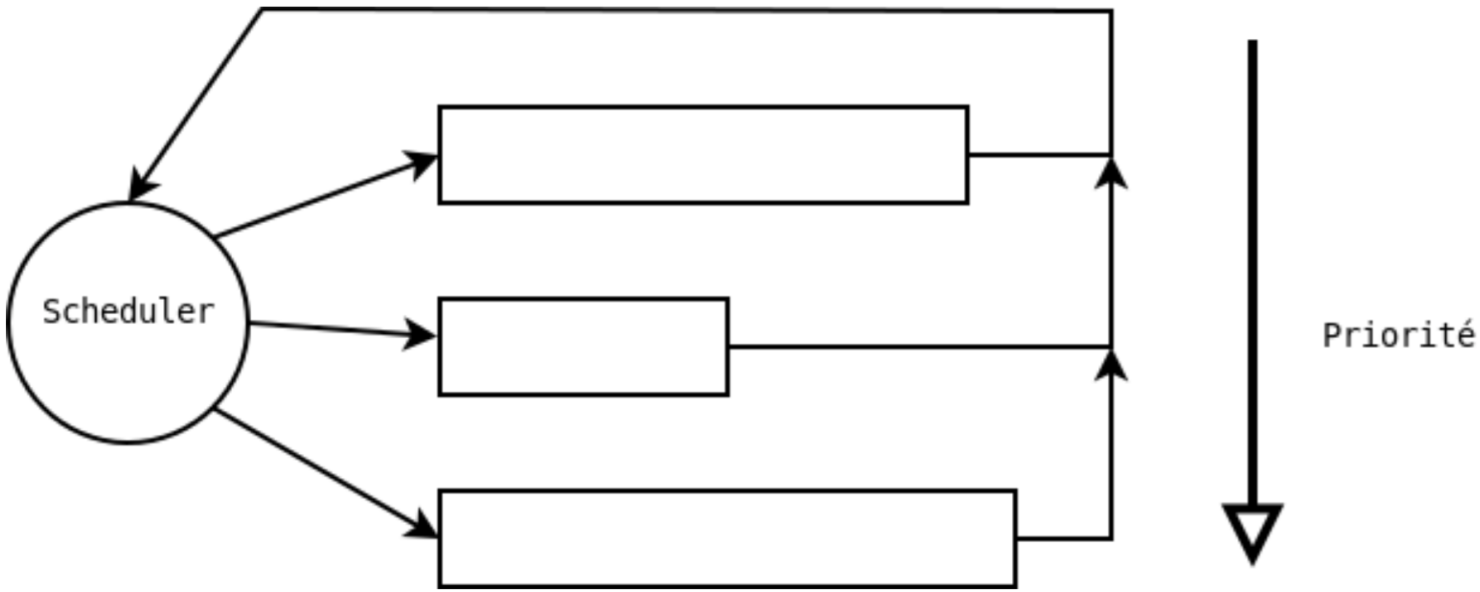
## Exemple d' Algorithme non-préemptif: FCFS

- FCFS : First Come, First Served
- Algorithme d'ordonnancement très simple mais peu rentable.
- Les processus longs sont favorisés par rapport aux processus courts.
- Gestion non-préemptif car rien n'est prévu par le système d'exploitation pour déloger le processus du CPU, ou favoriser un processus dans la file.

## Exemple d' Algorithme non-préemptif: Scheduler par priorité



## Exemple d' Algorithme non-préemptif: Scheduler par priorité




En Java, un thread a la priorité de son père. Il est possible de la changer: **setPriority (int prio)**.

## Exemple d' Algorithme préemptif: round-robin

- Basé sur: un quantum de temps et une file d'attente circulaire des threads.
- Soit le thread actif rend le fil d'exécution avant la fin de sa tranche de temps, soit il est préempté. Puis, il est placé en fin de liste.
- Chaque nouvel arrivant est placé à la fin de la file.
- Un thread obtiendra le processeur au bout de $(n-1) \times T$, où n est le nombre de threads et T est la durée du quantum de temps.

## Exemple d' Algorithme préemptif: round-robin

- Basé sur: un quantum de temps et une file d'attente circulaire des threads.
- Soit le thread actif rend le fil d'exécution avant la fin de sa tranche de temps, soit il est préempté. Puis, il est placé en fin de liste.
- Chaque nouvel arrivant est placé à la fin de la file.
- Un thread obtiendra le processeur au bout de $(n-1) \times T$, où n est le nombre de threads et T est la durée du quantum de temps.

**Remarque:** Attention au choix du quantum.

## Scheduler en Java

- Le choix en Java a été de ne pas faire un choix. Java ne définit que quelques règles sur la nature des threads et la façon de les ordonner, afin de garder son aspect multi-plateformes.

## Scheduler en Java

- Le choix en Java a été de ne pas faire un choix. Java ne définit que quelques règles sur la nature des threads et la façon de les ordonner, afin de garder son aspect multi-plateformes.
- Si l'on désire être réellement portable au sens de la spécification Java, il faut que les programmes puissent fonctionner, ceci demande de faire des appels à la méthode yield.

## Scheduler en Java

- Le choix en Java a été de ne pas faire un choix. Java ne définit que quelques règles sur la nature des threads et la façon de les ordonner, afin de garder son aspect multi-plateformes.
- Si l'on désire être réellement portable au sens de la spécification Java, il faut que les programmes puissent fonctionner, ceci demande de faire des appels à la méthode yield.
- Pourtant, une implémentation préemptive est favorisée, ce qui permet un mapping des threads sur les threads du SE hôte.

## Interface Future

- Certaines applications nécessitent de traiter des ensembles de tâches sur différents threads, souvent à des fins de performances.
- L'interface **Future** permet d'attendre ou de manipuler une tâche encore en cours d'exécution et dont on attend le résultat.
- **get()** retourne le résultat de la tâche et reste bloquante si le résultat n'est pas encore calculé.
- **cancel(), isCancelled(), isDone()**

## Pool de threads

- Pour éviter de devoir créer un thread à chaque fois qu'une nouvelle tâche doit être exécutée, par exemple dans le cas classique d'un serveur web. On utilise dans ce cas un **pool de thread**, qui n'est rien d'autre qu'une collection de threads qui se nourrit de tâches disponibles dans une queue.

## Pool de threads

- Pour éviter de devoir créer un thread à chaque fois qu'une nouvelle tâche doit être exécutée, par exemple dans le cas classique d'un serveur web. On utilise dans ce cas un **pool de thread**, qui n'est rien d'autre qu'une collection de threads qui se nourrit de tâches disponibles dans une queue.
- L'interface Executor permet de spécifier les méthodes classiques d'une tâche et l'on peut ensuite décider de différentes politique d'exécutions des tâches en agissant sur des objets implémentant Executor.

## Pool de threads

Les politiques proposées dans Java 1.5 sont les suivantes:

1. **Excecutors.newCachedThreadPool()**: un pool de thread de taille non limité, réutilisant les threads déjà créés et en créant de nouveau au besoin. Après 60 secondes d'inactivité, le thread est détruit.
2. **Excecutors.newFixedThreadPool(int n)**: un pool de thread de taille n. Si un thread se termine prématurément, il est automatiquement remplacé.
3. **Excecutors.newSingleThreadExecutor()**: crée un seul thread ce qui garantit la séquentialité des tâches mises dans la queue de taille non bornée.