Programmation Concurrente

Principes et concepts
2024-2025

Chapitre8: Java concurrence

Java concurrence

Notion de concurrence "système" (processus)

– java.lang.Runtime, Process, ProcessBuilder

- Bien mieux géré au niveau de la machine virtuelle
  - Classes spécifiques: java.lang.Thread, Runnable...
  - Classes de service: java.util.concurrent.Executors...
  - Cohérence des valeurs (volatile et classes atomics), relativement au Java Memory Model (JMM, JSR 133)
- Mémoire "globale" vs "locale"
  - Protection des accès et organisation des threads, exclusion mutuelle...
- synchronized, moniteurs, wait()/notify(), Synchronizers, java.util.concurrent.locks, collections concurrentes

Les processus

Objet représentant une application qui s'exécute

- java.lang.Runtime
  - Objet de contrôle de l'environnement d'exécution Objet courant récupérable par Runtime.getRuntime()
  - D'autres méthodes: [total/free/max]Memory(), gc(), exit(), halt(), availableProcessors()...
  - exec() crée un nouveau processus
- java.lang.Process et ProcessBuilder
  - Objets de contrôle d'un (ensemble de) processus, ou commandes
  - Runtime.getRuntime().exec("cmd") crée un nouveau processus correspondant à l'exécution de la commande, et retourne un objet de la classe Process qui le représente

Les processus légers (threads)

Étant donnée une exécution de Java (une JVM)

- **un seul processus** (au sens système d'expl)
- disposer de **multiples** fils d'exécution (**threads**) internes
- possibilités de **contrôle** plus fin (priorité, interruption...)
- c'est la JVM qui assure l'**ordonnancement** (concurrence)
- espace **mémoire commun** entre les différents threads

– Deux instructions d'un même processus léger doivent en général respecter leur séquencement (sémantique)

– Deux instructions de deux processus légers distincts n'ont pas *a priori* d'ordre d'exécution à respecter (entre-eux)

La classe java.lang.Thread

Chaque instance de la classe Thread possède:

- un nom, [get/set]Name(), un identifiant
- une priorité, [get/set]Priority(),
- les threads de priorité haute sont exécutées + souvent
- trois constantes prédéfinies: [MIN / NORM / MAX]_PRIORITY
- un statut *daemon* (booléen), [is/set]Daemon()
- un groupe, de classe ThreadGroup, getThreadGroup()
- par défaut, même groupe que la thread qui l'a créée
- une cible, représentant le code que doit exécuter ce processus léger. Ce code est décrit par la méthode

```
public void run() {...}
```

qui par défaut ne fait rien (return;) dans la classe Thread

Threads et JVM

La Machine Virtuelle Java continue à exécuter des threads jusqu'à ce que:

- soit la méthode exit() de la classe Runtime soit appelée
- soit toutes les threads non marquées "*daemon*" soient terminées.

– on peut savoir si une thread est terminée via la méthode isAlive()

- Avant d'être exécutées, les threads doivent être créés: Thread t = new Thread(...);
- Au démarrage de la thread, par t.start();
- la JVM réserve et affecte l'espace mémoire nécessaire

avant d'appeler la méthode run() de la cible.

Le thread courant

```
public class ThreadExample {
        public static void main(String[] args)
                        throws InterruptedException {
                Thread t = Thread.currentThread();
                // Affiche caractéristiques de la thread
                courante
                System.out.println(t);
                // Lui donne un nouveau nom
                t.setName("Médor");
                System.out.println(t);
                // Rend le processus léger courant
                // inactif pendant 1 seconde
                Thread.sleep(1000);
                System.out.println("fin");
}
}
```

Deux façons pour spécifier run()

- Redéfinir la méthode run() de la classe Thread

–
```
class MyThread extends Thread  {
        public @Override
        void run() { /* code à exécuter*/ }
}
```

– Création et démarrage de la thread comme ceci:

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

- Implanter l'interface Runnable

–
```
class MyRunnable implements Runnable {
         public void run() { /* code à exécuter */ }
         }
```

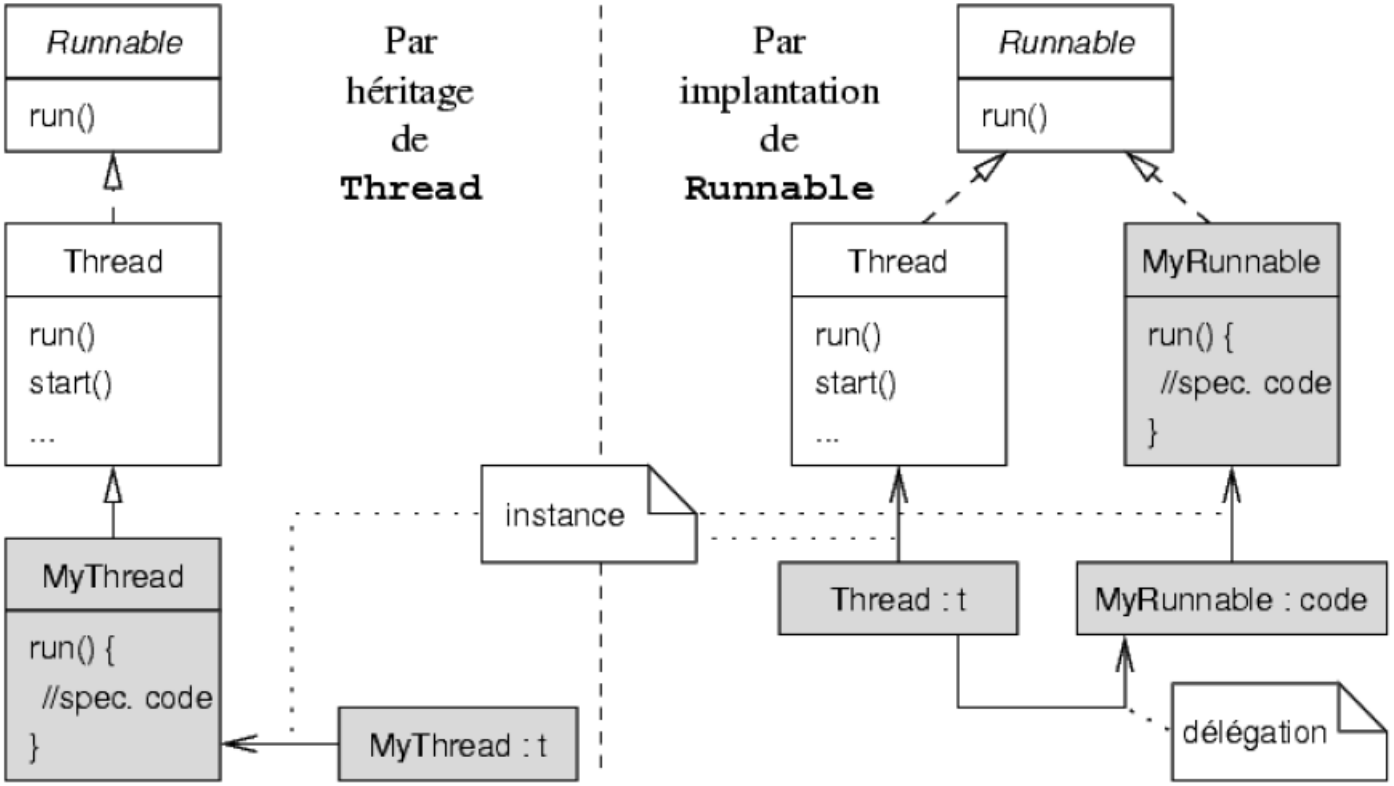
– Création et démarrage de la thread via un **objet cible**:

```
MyRunnable cible = new MyRunnable(); // objet cible
```

`Thread t = new Thread(cible);` puis `t.start();`

Comparaison des deux approches

- Pas d'héritage multiple de classes en Java: hériter d'une autre classe?
- Pouvoir faire exécuter un **même** Runnable à plusieurs threads



Par héritage de Thread

```
public class MyThread extends Thread {
  public void run() {
              for (int i=0; i<5; i++) {
                System.out.println("MyThread, en " + i);
                try {Thread.sleep(500);}
                catch (InterruptedException ie) {ie.printStackTrace();}
               }
              System.out.println("MyThread se termine");
}
}
public class Prog1 {
    public static void main(String[] args) throws InterruptedException {
         Thread t = new MyThread(); t.start();
         for (int i=0; i<5; i++) {
              System.out.println("Initial, en " + i);
              Thread.sleep(300);
                   }
                   System.out.println("Initial se termine");
}}
```

Par implantation de Runnable

```
public class MyRunnable implements Runnable{
  public void run() {
              for (int i=0; i<5; i++) {
                System.out.println("MyRunnable, en " + i);
                try { Thread.sleep(500); }
                catch (InterruptedException ie) {ie.printStackTrace();}
               }
              System.out.println("MyRunnable se termine");
}
}
public class Prog2 {
    public static void main(String[] args) throws InterruptedException {
         MyRunnable cible = new MyRunnable();
         Thread t = new Thread(cible); t.start();
                    for (int i=0; i<5; i++) {
              System.out.println("Initial, en " + i);
              Thread.sleep(300);
                   }
                   System.out.println("Initial se termine");
}}
```

Les méthodes

- static Thread currentThread();
- static void sleep(long millis);
- static void yield(); // passe la main
- static boolean interrupted(); // clear status
- void run();
- void start();
- void interrupt();
- boolean isInterrupted(); // keep status
- void join(); // thread.join() attend la fin de thread
- InterruptedException // clear status

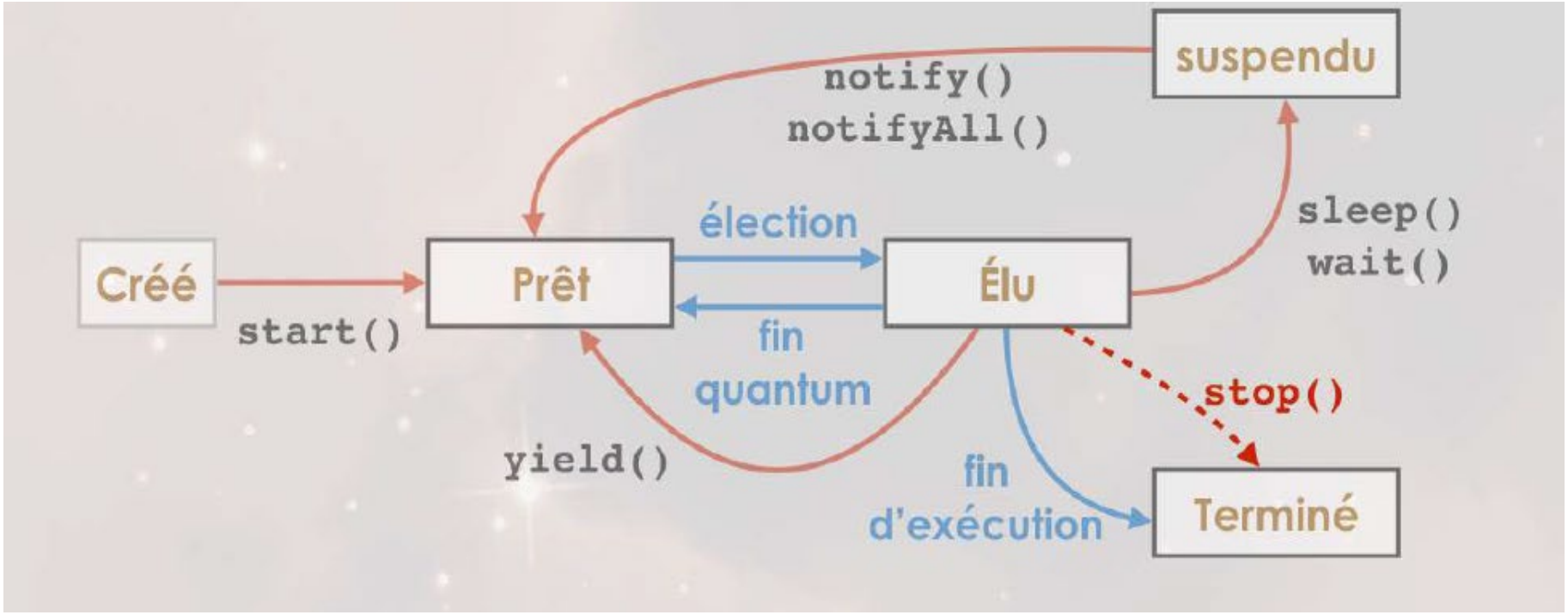
Cycle de vie d'un processus léger

- Création de l'objet contrôleur: t = new Thread(...)
- Allocation des ressources: t.start()
- Début d'exécution de run()
  - [éventuelles] suspensions temp. d'exéc: Thread.sleep()
  - [éventuels] relâchements voulus du proc. : Thread.yield()
  - peut disposer du processeur et s'exécuter
  - peut attendre le proc. ou une ressource pour s'exécuter
- Fin d'exécution de run()
- Ramasse-miettes sur l'objet de contrôle

L'accès au processeur

- Différents états possibles d'une thread
  - exécute son code cible (elle a accès au processeur)
  - attend l'accès au processeur (mais pourrait exécuter)
  - attend un événement particulier (pour pouvoir exécuter)
- L'exécution de la cible peut libérer le processeur
  - si elle exécute un yield() (demande explicite)
  - si elle exécute une méthode bloquante (sleep(), wait()...)
- Sinon, c'est l'ordonnanceur de la JVM qui répartit l'accès des threads au processeur.
  - utilisation des éventuelles priorités

Le cycle complet d'une thread



Terminaison d'une thread

- Les méthodes stop(), suspend(), resume() sont dépréciées
  - Risquent de laisser le programme dans un "sale" état !
- La méthode destroy() n'est pas implantée
  - Spécification trop brutale: l'oublier
- Seule manière: terminer de manière **douce**...
- Une thread se termine normalement lorsqu'elle a terminé d'exécuter sa méthode run()
  - obliger **proprement** à terminer cette méthode

Interrompre une thread

- La méthode interrupt() appelée sur une thread t
  - Positionne un « statut d'interruption »
  - Si t est en attente parce qu'elle exécute un wait(), un join() ou un sleep(), alors ce statut est réinitialisé et la thread reçoit une InterruptedException
  - Si t est en attente I/O sur un canal interruptible (java.nio.channels.InterruptibleChannel), alors ce canal est fermé, le statut reste positionné et la thread reçoit une ClosedByInterruptException *(on y reviendra)*
- Le statut d'interruption ne peut être consulté que de deux manières, par des méthodes (pas de champ)

Exemple d'interruption

```
import java.io.*;
public class InterruptionExample implements Runnable {
    private int id;
    public InterruptionExample(int id) {
        this.id = id;
         }
         public void run() {
        int i = 0;
        while (!Thread.interrupted()) {
            System.out.println(i + "i° exécution de " + id);
            i++;
        }
        System.out.println("Fin d'exéc. du code " + id);
        // L'appel à interrupted() a réinitialisé
        // le statut d'interruption
        System.out.println(Thread.currentThread()
        .isInterrupted()); // Affiche: false
}}
```

L’usage de interrupt

- L’usage de interrupt() est délicat
  - Il ne faut pas laisser le programme dans un état incohérent
  - Il faut contrôler «quand» meurent les threads
- Deux approches
  - L’environnement s’en charge(Posix)
  - Le développeur s’en charge(Java)