Concurrence
Interruption et exceptions

Exceptions

En Java, il y a deux façon de sortir d'un appel de méthode

- Par un appel à return (ou un return implicite)
- Par la levée d'un exception

Il existe deux sortes d'exceptions

- Les exceptions non checkés

  NullPointerException, ArrayIndexOutOfBoundsException

  les erreurs du programmeur

- Les exceptions checkés
  - IOException, InterruptedException

    les erreurs du à une action extérieur
    (sur lesquelles, on doit reprendre)

Appels Blockants

Un appel système peut être

- Blockant

  si lorsqu'il n'a pas accès à une ressource met la thread courante en attente jusqu'à ce que la resource arrive

- Non blockant
  - Si lorsqu'il n'a pas accès à une resource, le système renvoie un code indiquant qu'il n'a pas accès à la ressource

Appels Blockant

Les appels à la VM

- Thread.sleep(), Lock.lockInterruptibly() ou Object.wait()

sont aussi des appels blockant et fonctionne de la même façon en Java

Arrêter un Thread

- Il n’est pas possible d’arrêter/tuer un Thread en Java
  - Historiquement, il y avait une méthode destroy() mais elle n’a jamais marché
    - Elle a été retiré en Java 9
- La seule façon d’arrêter une Thread est de lui demandée gentillement

Mécanisme de signalisation

Une thread peut envoyer un signal à une autre thread

```
autreThread.interrupt();
```

L'autre thread reçoit le signal de 2 façon différentes

- Si la thread est dans un appel bloquant
  - l'exception InteruptedException est levée
- Si la thread n'est pas dans un appel bloquant
  - Un boolean est positionné (testable avec Thread.interrupted())

Exemple

Avec un appel bloquant

```
Thread t = new Thread(() → {

    Thread.sleep(5_000);

});
t.start();
…
t.interrupt();
```

Ce programme ne compile pas car Thread.sleep() peut lever InterruptedException qui doit être traité (checked exception)

Exemple (2)

Avec un appel bloquant

```
Thread t = new Thread(() → {
    try {
      Thread.sleep(5_000);
    } catch(InterruptedException e) {
      // ici je doit ABSOLUMENT traitée l'exception
    }
});
t.start();
…
t.interrupt();
```

Attention, un catch qui ne fait rien, cela veut dire que l'on ne pourra jamais arrêter la thread.

Exemple (3)

Il y a deux façon de traiter les checked exceptions

- Sortir du main / runnable.run()
- Lever une unchecked exception pour sortir du main / runnable

```
Thread t = new Thread(() → {
    try {
      Thread.sleep(5_000);
    } catch(InterruptedException e) {
      return;      // ou throw new AssertionError(e);
    }
});
t.start();
…
t.interrupt();
```

Et si pas d’appel blockant

Sans un appel bloquant

```
Thread t = new Thread(() → {
   for(long l = 0; l < 1_000_000_000L; l++) {
      // calcul sans appel bloquant
   }


});
t.start();
…
t.interrupt();
```

Dans ce cas, le programme ne sait pas qu’il a été interrompu

Thread.interrupted()

Sans un appel bloquant

```
Thread t = new Thread(() → {
   for(long l = 0; l < 1_000_000_000L && !Thread.interrupted(); l++) {
      // calcul sans appel bloquant
   }
});
t.start();
…
t.interrupt();
```

Il faut tester le status d'interruption.

Note: Thread.interrupted() remet le status d'interruption à false (c'est la seule façon de le mettre à false)

Et si on a un mix

Avec des appels mais pas tout de suite

```
Thread t = new Thread(() → {
   for(long l = 0; l < 1_000_000_000L; l++) {
      // calcul sans appel bloquant
   }
   try {
     Thread.sleep(5_000);
   } catch(InterruptedException e) {
     return;
   }
});
t.start();
…
t.interrupt();
```

Si le status d'interruption est vrai, alors un appel bloquant ultérieur lève InterruptedException

Mais cela veut dire attentre la fin du calcul :(

S’interrompre soit-même

Permet de sortir de la boucle d’une seule façon, pratique si on doit librérer des ressources, etc

```
Thread t = new Thread(() → {
   for(long i = 0; l < 1_000 && !Thread.interrupted(); l++) {
      // calcul long sans appel bloquant

      try {
        Thread.sleep(5_000);
      } catch(InterruptedException e) {
        Thread.currentThread().interrupt();
        continue;
      }
   }
   // libérer les ressources ici !
});
t.start();
…
t.interrupt();
```

Interrupt() et convention

Habituellement, appeler interrupt() sur un Thread veut dire que l'on demande à ce qu'il s'arrête

Mais ce n'est qu'une convention, le code de la Thread peut faire ce que bon lui semble (pourvu que cela soit écrit dans la javadoc)

Ne **rien** faire (ou printStackTrace), est **jamais** la bonne solution

Appel qui lève IOException

Les appels bloquants (Reader.read, InputStream.write)

- qui lève IOException ne lève pas InterruptedException mais IOInterruptedException (qui est une sous-classe de IOException)

donc de la même façon, lorsque l'on recoit une IOException, il faut tout fermer et arrêter la Thread

- Ou pour un serveur, arrêter la connection avec le client

Synchronized

- Un bloc synchronized est bloquant si une thread a déjà pris le jeton associé au moniteur
  - Il n'y a pas d'InterruptedException a attrapé car normalement, on fait peu de chose dans un bloc synchronized, donc les Threads attendent pas longtemps

- Si on veut permettre l'interruption d'une section critique, il faut utiliser des ReentrantLock (et la méthode lockInterruptibly())