Threads

Introduction aux threads ¶ §

Qu'est-ce que c'est ?

- Java propose un mécanisme pour exécuter parallèlement plusieurs tâches de manière indépendante ou synchonisée au sein d'un même processus.
- Un **thread** est une exécution particulière et indépendante dans l'espace mémoire alloué à un processus.
- Toutes les méthodes s'exécutent dans au moins un thread; la méthode `main` s'exécute dans un thread principal. On peut créer de nouveau threads.
- Consultez la documentation de `java.lang.Object` et `java.lang.Thread` ainsi que le tutoriel sur la programmation concurrente.

Comment créer un thread ?

Première façon de faire: dériver `java.lang.Thread`.

```
public classe MonPremierThread extends Thread {
  public void run() { ... }
}
```

L'appel à la méthode `start` (héritée) lance l'exécution de la méthode `run`. L'exécution du thread se termine au retour de la méthode `run`.

```
MonPremierThread t = new MonPremierThread();
monPremierThread.start();
```

Comment éviter l'héritage ?

L'héritage de `java.lang.Thread` est contraignant car il empêche tout autre héritage. Il existe une seconde façon de faire: implémenter l'interface `java.lang.Runnable`.

```
public classe MaClasseRunnable implements Runnable {
  public void run() { ... }
}
```

On passe une instance de la classe implémentant `Runnable` au constructeur de `Thread`.

```
MaClasseRunnable r = new MaClasseRunnable();
```

```
Thread t = new Thread( r );
t.start();
```

Méthodes de la classe `Thread`

- statique
  - `currentThread()`: référence vers le thread courant
- non statique
  - `getName()`
  - `start()`
  - `isAlive()`
  - `sleep()`: mise en attente pour une durée donnée.

Synchronisation physique

Etats d'un thread

Un thread peut être:

- en *exécution*
- *prêt* à s'exécuter, en attente d'un processeur libre.
- *bloqué*, en attente d'une ressource partagée pour poursuivre son exécution.

Les threads utilisant la même ressource partagée doivent être synchronisés.

Partage de la mémoire

Chaque instance de `Thread`, chaque instance d'une classe `Runnable` possède ses propres champs. Pour partager une donnée entre plusieurs threads:

- soit on travaille avec un champs statique de la classe implémentant l'interface `Runnable` (partagée par toutes les instances de cette classe)
- soit on travaille avec des références vers un même objet partagé. C'est ce qui est conseillé, car c'est plus simple de travailler avec un objet.

Ex.1. Compteur (10 min)

- Téléchargez la classe **`EvenCounter`**. Que fait-elle ?
- Ecrivez la classe `EvenCounterTest` dans laquelle vous instanciez un seul objet de la classe `EvenCounter`, que vous exécutez dans deux threads.
- Compilez et exécutez plusieurs fois. Que se passe-t-il ? Pourquoi ?
- Ajoutez le mot-clef `synchronized` à la méthode `toNextEven`. Que se passe-t-il ?

```
private synchronized void toNextEven() {
```

Ex.2. Tableaux de threads (10 min)

- Téléchargez la classe **`Piscine`**. Que fait-elle ?
- Téléchargez la classe **`Baigneur`**. Que fait-elle ?
- Ecrivez une classe `BaigneursTest` qui lance des threads opérant sur 150 instances de la classe `Baigneur`, chacune connaissant un seul objet de type `Piscine`:

```
Piscine piscine = new Piscine();    //la piscine
int n = 150;
Thread[] baigneurs = new Thread[n];
for (int i = 0; i < n; i++)         //les baigneurs
    baigneurs[i] = new Thread( new Baigneur(piscine, 5) );
```

- Compilez, puis exécutez plusieurs fois. Est-ce que ça fonctionne ?

Ex.3. Accès concurrents (10 min)

- Dans la classe `Piscine`, ajoutez des sections critiques avec la construction suivante:

```
synchronized (this) {
   ...
}
```

L'objet entre parenthèse est utilisé de manière exclusive par le thread courant. L'exécution des autres threads est bloquée jusqu'à ce que le thread courant exécute la dernière instruction du bloc.

Ce qu'il faut retenir

Quand plusieurs threads partagent des données, il peut y avoir *interférence* (deux exécutions d'une même méthode sont entrelacées) ou *incohérence* (les appels de différentes méthodes d'un même objet sont entrelacés).

Pour éviter ces problèmes, on peut définir des **sections critiques** avec le mot-clef `synchronized`.

- L'objet dont une méthode qualifiée `synchronized` est exécutée par un thread n'est plus disponible pour les autres threads.
- Le bloc `synchronized` permet d'utiliser de manière exclusive un objet par le thread courant.

Dans les deux cas, la synchronisation porte sur *un objet particulier*.

Synchronisation temporelle

Méthodes

- Méthode de `java.lang.Thread`:
  - `join` : met en attente le thread courant, jusqu'à ce que le thread dont on appelle la méthode `join` meurt.
- Méthodes de `java.lang.Object`:
  - `wait`: met en attente le thread courant sur l'objet auquel la requête est adressée, jusqu'à ce qu'il soit réveillé ou interrompu par un autre thread (ou jusqu'à ce qu'une durée donnée se soit écoulée).
  - `notify`: réveille le thread qui est en attente sur l'objet dont on appelle la méthode `notify` (s'il y en a plusieurs, l'un d'eux est choisi arbitrairement).
  - `notifyAll`: réveille tous les threads en attente sur l'objet.

Problème producteur/consommateur

- Imaginons un producteur; il produit des objets et les entreprose. Mais il n'y a qu'une seule place.
- Imaginons un consommateur; il retire l'objet entreposé.
- Comment synchroniser leurs actions afin que le producteur n'essaie d'entreproser un nouvel objet que lorsque la place est libre et que le consommateur n'essaie de retirer un nouvel objet que lorsqu'un objet est disponible ?

Ex.4. Wait/Notify (20 min)

- Téléchargez cette **`archive`**.
- Que fait la classe `ProducerConsumerTest` ? Compilez et exécutez. Que se passe-t-il ?
- Ecrivez une classe `SyncCubbyHole`, qui étend `CubbyHole` et qui redéfinit les méthodes `get` et `put` en les marquant `synchronized` et en appelant les méthodes `wait` et `notify`.

Ce qu'il faut retenir

- Tous les objets peuvent mettre en attente le thread courant avec `wait`.
- Tous les objets peuvent réveiller le(s) thread(s) bloqué(s) par eux, avec `notify` et `notifyAll`.

Pour aller plus loin

Fabrique de threads

Le package `java.util.concurrent` contient une classe `Executors` fabriquant:

- un thread avec `newSingleThreadExecutor()`
- un pool de threads en appelant `newFixedThreadPool()`

Ces méthodes renvoient en fait un objet de type `ExecutorService`, sous-type de `Executor`. Autrement dit, un objet issu d'une classe implémentant l'interface `ExecutorService`, dérivant l'interface `Executor`.

Executor

Les objets de type `Executor` possèdent une méthode `execute()` qui crée, puis démarre un thread.

Si `e` est un objet de type `Executor` et si `r` est un objet de type `Runnable`, alors ces codes sont équivalents:

```
Thread t = new Thread(r);
t.start();
```

```
e.execute(r);
```

A la maison. Pool de threads (10 min)

- Ecrivez une classe `BaigneursTest2` qui, au lieu de manipuler un tableau de threads comme dans `BaigneursTest`, utilise le pool de threads renvoyé par la méthode `newFixedThreadPool()` de `Executors`.
- Appelez la méthode `shutdown()` pour finir l'exécution des threads et ne plus attendre de nouvelles tâches.
- Testez avec un nombre de threads égal à 150, puis 50, puis 3.