Développement en JAVA

Threads et parallélisme

Agenda

- Notions de bases sur les threads
- La classe Thread
- La classe Runnable
- Atomicité
- Concurrence

Notions de base

- La plus part des OS contemporains sont multi tache
- Il peuvent exécuter plusieurs programme en même temps, on parle de **processus**
- A un instant t il ne peut y avoir plus de processus en cours d'exécution que le matériel possède de processeur
- Le parallélisme est donc « simulé » par l'OS, on parle d'**ordonnancement**

Notions de base

- La plus part des OS sont **préemptif**: c'est l'OS qui décide quel programme à le droit de s'exécuter sur le ou les processeurs, et pour combien de temps
- Le programmeur ne peut pas savoir quand l'exécution du programme sera interrompu par l'OS

Notions de thread

- Un thread est un « processus léger »
- Il est créé par et est attaché à un processus
- Un processus peut créer plusieurs threads
- Les threads sont ordonnancés de manière concurrente, de la même manière que les processus
- On utilise les threads pour gérer la notion de **parallélisme** à l'intérieur d'un processus
- Les threads d'un même processus **partagent le même espace mémoire** (même heap)

La classe Thread

- La classe `java.lang.Thread` représente un thread du point de vu JVM
- Elle permet de créer et démarrer un thread
- Elle se construit en général à l'aide d'un objet héritant de l'interface `Runnable`

```
Thread t1 = new Thread(unObjetRunnable);
Thread t2 = new Thread(unObjetRunnable);
t1.start();
t2.start();
// A présente deux threads s'éxecutent en parallèle
```

L'interface Runnable

- L'interface Runnable n'expose qu'une seule méthode `public void run()`
- On implémente dans cette méthode le code qui sera exécuté au démarrage du thread
- On peut bien sur avoir accès à l'ensemble des attributs de l'objet

```
public class Parser implements Runnable{
  public void run(){
    // Le code du thread
  }
}
```

La méthode sleep()

- La méthode statique `Thread.sleep(int millisecondes)` permet de mettre en pause un thread pendant X millisecondes
- Elle est notamment utile pour simuler des temps de réponse ou des situation d'ordonnancement
- C'est une méthode statique, elle donc accessible depuis n'importe où dans votre programme

```
// Arrêt du thread pendant 10 secondes
Thread.sleep(10000);
```

Notion d’atomicité

- Une opération atomique est une opération ne pouvant pas être interrompue par l’ordonnanceur
- Très peu d’instructions sont atomiques en Java
- L’utilisation de variables partagés en Thread doit donc être implémenté prudemment
- Deux modifications concurrentes à une même variable peuvent aboutir à des comportement incertains

Notion d'atomicité

```
private static int nbThread = 0;

public void run(){
  // Non thread safe, car l'opération ++ n'est pas atomique
  nbTread++;
}
```

- Les classes `AtomicInteger`, `AtomicBoolean` etc… Offrent un moyen efficace de s'assurer de l'atomicité d'une opération

```
private static AtomicInteger nbThread = 0;

public void run(){
  // Thread Safe
  nbTread.addAndGet(1);
}
```

Le mot clé syncrhonized

- Le mot clé synchronized permet de sécuriser une portion de code
- Il permet de s'assurer que seul un Thread est capable d'être présent dans le bloc en même temps
- On peut l'utiliser directement dans la signature d'une méthode

```
private static int nbThread = 0;

public void synchronized increment(){
  nbTread++;
}
```

Le mot clé synchronized

- On peut l'utiliser directement dans la signature d'une méthode

```
private static int nbThread = 0;

public void run(){
  synchronized(this){
    nbTread++;
  }
}
```

DeadLock

- Les portions de code synchronized permettent de mettre en place des verrous sur certaines portion du code
- Si un thread reste bloqué dans une portion synchronized, il peut bloquer tout le reste de l'application
- On portera attention à réduire à son minimum la taille de la portion de code sous la portée d'un mot clé syncrhonized
- On évitera en particulier d'y placer toute opération d'I/O