Annexe : introduction aux threads

Introduction

Il est souvent nécessaire qu'un seul programme soit capable d'exécuter plusieurs tâches indépendantes et simultanées (l'affichage de l'heure et la saisie, le calcul et le clic du bouton, le déplacement des deux personnages). De tels programmes sont dits "**multithread**" et une des tâches exécutées par ces programmes est appelée "**thread**".

> ***définition*** : **un programme multithread** est un programme capable d'exécuter en même temps plusieurs séquences d'instructions, chacune disposant de son propre contrôle de déroulement (i.e. son propre point d'entrée, sa propre logique, son propre point d'arrêt), tout en partageant un espace mémoire commun, ce qui permet de partager des variables.

Il est possible de se demander comment, sur un ordinateur, faire tourner plus d'un thread. Deux cas sont possibles :

- sur un système monoprocesseur, seul un thread s'exécute à un instant donné, mais un mécanisme géré par le système d'exploitation (appelé scheduler ou planificateur) permet de passer rapidement de l'exécution d'un thread à un autre pour donner l'illusion qu'ils s'exécutent en même temps. On parle de simultanéité logique.
- sur un système multiprocesseur, plusieurs threads s'exécutent effectivement en même temps. La simultanéité est physique.

Java par le biais de sa machine virtuelle gère le multithreading.

I. CREATION DE THREADS

La gestion des threads (de base) est effectuée à l'aide des trois classes suivantes :

- l'interface **java.lang.Runnable : elle sert à définir le travail exécuté par le thread**
- la classe **java.lang.Thread, implémentant Runnable : elle contrôle le thread (par exemple, elle l'exécute)**
- la classe java.lang.Object : **elle permet de gérer des problèmes de verrouillage d'objet** (donc de synchroniser plusieurs threads)

Il est également important de noter qu'en fait, à chaque exécution d'applications, plusieurs threads peuvent être lancés :

- **le traitement de main, qui forme un thread à part entière, lancé par la JVM, et qui fournit le traitement à partir duquel tous les autres threads sont lancés,**
- les threads gérés par les interfaces graphiques,
- les threads fabriqués par les programmeurs, à l'aide de la classe Thread.

Deux techniques, basées sur Thread et Runnable, existent donc pour la création d'objet thread.

a) la dérivation de java.lang.Thread

La première méthode consiste à créer une dérivée de la classe Thread et à redéfinir la méthode run(), elle-même héritée de l'interface Runnable, pour spécifier le point d'entrée et le traitement effectué par le thread.

```
class Work extends Thread {
  public Work() {.
..} // constructeur
 ...
 public void run() {
 // Ici ce que fait le thread : boucle infinie ou pas  }
}
```

L'exécution du thread est donc lancée par les instructions suivantes :

```
...
Work work = new Work(); // Création du thread t
work.start(); // démarre le thread et execute work. run()
```

Remarques générales:

- quand la méthode run se termine, le thread est terminé et ne pourra jamais être redémarré ou réutilisé (le lancer deux fois déclenche une exception de nature :java.lang.IllegalThreadStateException)

- il existe une méthode statique retournant une référence sur le thread en cours d'exécution :
  static Thread currentThread()

- la classe Thread possède un certain nombre de données membres importantes : un nom (String) accessible par getName() et setName(), une priorité (int) : chaque thread a un niveau de priorité entre 1 (Thread.MIN_PRIORITY) et 10 (Thread.Max_PRIORITY) accessible par setPriority() et int getPriority().

- un statut d'interruption : il s'agit d'un booléen interne qui peut être utilisé de multiples manières et notamment pour stopper un thread en cours d'exécution (à partir d'un autre thread ou de main).

  Agir sur ce booléen se fait à l'aide d'une fonction membre appelée void interrupt(). Cette fonction a deux effets :

  - si le thread est en attente (par l'utilisation de sleep par exemple, qui permet d'interrompre un thread pendant un certain laps de temps, mais aussi par wait ou autre), une exception de nature InterruptedException est émise et le statut d'interruption est placé à false,
  - si le thread exécute n'importe quoi d'autre, aucune exception n'est émise et le statut d'interruption est mis à true.

  Pour connaître le statut d'erreur, il suffit d'utiliser la méthode boolean isInterrupted().

  Connaissant cet état, il est alors très facile de déterminer une boucle de traitement :

```
try {
        while (!isInterrupted()) {
                ...
                Thread.sleep(1000);
        }
} catch (InterruptedException e) {
        System.out.println("exception émise par sleep");
}
```

Exercice 1 :

a) Créer et lancer un thread permettant d'afficher l'heure toutes les secondes.
b) Créer deux threads affichant un message toutes les x millisecondes (observer l'entrelacement des instructions)

b) l'implémentation de java.lang.Runnable

L'autre possibilité consiste à implémenter l'interface **Runnable** et à redéfinir la méthode run(), qui contient le corps des instructions du thread.

```
class Work implements RunnableThread {
  Work () {
.} // constructeur
  ...
  public void run()  {
  // Ici ce que fait le processus : boucle infinie ou pas   }
}
```

L'exécution se fait donc par :

```
...
Work work = new Work ();
Thread thread = new Thread(work);
...
thread.start(); // démarre un thread qui execute work. run()
```

Exercice 2 :

Transformer l'un des threads précédents en implémentation de Runnable.

II. TRAITEMENTS BASIQUES SUR THREADS

a) la jointure entre threads

Il est parfois nécessaire qu'un thread attende la fin d'un autre thread pour pouvoir continuer à s'exécuter :

- avant de déclencher une interrogation, il faut avoir chargé un fichier complet
- un calcul de prix dépend de deux calculs (l'un pour le prix HT d'une facture, l'autre permettant de déterminer une réduction à partir d'un code de client en provenance d'une base de données)
- …

Il est dans ce cas nécessaire que le premier thread attende la fin de l'exécution du second avant de continuer. La solution consiste à utiliser la méthode join, appliqué au thread dont on attend la fin de l'exécution.

Syntaxe :

thread1.join() attend la fin de l'exécution de thread1

Par exemple :

```
double billCompute() {
        …
        thread1.start();        // lancement du thread de calcul du prix
        thread2.start();        // lancement du thread de calcul de la réduction
        thread1.join();         // main attend la fin de l'exécution de tache1
        thread2.join();         // main attend la fin de l'exécution de tache2

        return thread1.getPrice() + thread2.getReduction();
}
```

Exercice 3

a) Construire deux fichiers de réels aléatoires conséquents (un million de réels environ),
b) Construire une fonction retournant la somme des valeurs des deux fichiers calculées par deux threads ayant pour but d'en additionner les valeurs

b) Passer la main

Il est aussi possible de forcer un thread à passer la main à un autre thread de priorité égale ou supérieure, lorsque le programmeur estime que celui-ci a suffisamment travaillé ou bien qu'il souhaite qu'un autre thread puisse s'exécuter (important dans le mode coopératif).

Le traitement a utiliser ici est disponible au travers de la méthode yield.

```
this.yield();
```

Par contre, il n'est pas possible de choisir celui qui va être exécuté.

S'il n'y a pas de thread de même priorité ou de priorité supérieure, alors l'appel à la méthode yield est ignoré.

c) Stopper un thread

Si l'on ne souhaite pas travailler avec le booléen interrupt, il est possible de stopper un thread en gérant soi-même un booléen comme par exemple dans :

```
class TryStop extends Thread {
        boolean done= false;

        public void run() {
         while(!done) {
           System.out.println("EssaiStop en cours d'exécution");
           try{
                sleep(720);
           } catch(InterruptedException e) {
               // rien à faire
            }
          }
         System.out.prinln("Termine");
        }

       public void end() {
        done=true; }
       }
}
```

d) Savoir si un thread est en cours d'exécution

La méthode isAlive() retourne *true si le thread a été démarré (start) et n'est pas arrêté :*

- si isAlive() retourne false, le thread vient d'être créé (état né) ou est mort (terminé)
- si isAlive() retourne true, le thread est en cours d'exécution
- il n'est pas possible de distinguer les états né/terminé et exécution (prêt/en cours)

e) Méthodes dépréciées

Ne jamais utiliser stop, ni destroy, ni resume, ni suspend; ces fonctions sont dépréciées et amènent les threads à des situations de bloquage.

QUELQUES MEMBRES DE LA CLASSE THREAD

METHODES	***DESCRIPTIONS***
public Thread()	Constructeur par défaut
public Thread(String name)	Constructeur initialisation (fixe le nom du thread)
public Thread(Runnable cible)	Constructeur initialisation avec objet Runnable
public Thread(Runnable cible, String name)	Idem + fixe le nom du thread
public final String getName()	Renvoie le nom d'un thread Le nom par défaut : « Thread-1 »
public final void setName(String name)	Fixe le nom d'un thread
public final boolean isAlive()	Renvoie true si le thread est actif, false sinon

MEMBRES	***DESCRIPTIONS***
public static final int MAX_PRIORITY	Niveau maximum de priorité d'un thread =10
public static final int NORM_PRIORITY	Niveau normal de priorité d'un thread =5 (valeur par défaut)
public static final int MIN_PRIORITY	Niveau minimum de priorité d'un thread =1