TD n° 2- Gestionnaires de tâches

Dans ce TD, nous allons continuer l'étude des Threads en *Java*. On étudie ici, comment des gestionnaires différents de celui du système peuvent être utilisés pour exécuter des tâches (attention : qui ne sont pas forcement des *threads*, mais les gestionnaires peuvent utliser les *threads* pour les réaliser).

Rappel de cours : Les classes `Executor` et `ExecutorService` sont des boîtes á outils (des interfaces) pour la création et la gestion de groupe de tâches. La gestion des tâches est déléguée à un `Executor` en lui passant la tâche par sa méthode `execute(Runnable tache)`. Dans ce cas, les tâches sont crées et exécutees par le gesionnaire (souvent sous forme des threads).

Plusieurs implémentations de ces interfaces existent. Certaines sont offerte par la classe `Executors`. Par exemple la méthode `static ExecutorService newFixedThreadPool(int nbtaches)` du fabrique `Executors` retourne un objet qui implémente l'interface `ExecutorService` . Ce gestionnaire permet de gérer (exécuter) un ensemble de tâches avec une limitation sur le nombre de tâches actives. Le nombre de tâches est le nombre maximum d'exécutions simultanées. Les tâches supplémentaires sont placées dans une file d'attente et attendent qu'une tâche active se termine.

En général, pour pouvoir exécuter des tâches en utilisant un gestionnaire particulier existant, il faut

— créer un gestionnaire de tâches (un objet de la classe `Executor` ou `ExecutorService`[1]);
— créer une instance pour chaque tâche á exécuter en implémentant l'interface `Runnable`;
— rendre pour l'exécution chacune des tâches en appelant la méthode `execute()` du gestionnaire de tâches. Attention : c'est le gestionnaire qui s'occupe de la création des objets (threads) nécessaires et de l'exécution.

Exercice 1. *Ecriture d'un gestionnaire `Executor`*

L'interface `Executor` est très simple. Ses implémentations permettent de transmettre des `Runnables` pour créer et exécuter des `Threads`. Il est toujours possible d'ajouter des services à cette première version très simple. (`ExecutorService` est aussi une interface qui enrichie les services par héritage. Etudiez la documetation.)

1. Dans cette exercice, on demande *l'écriture* d'un `Executor` instanciable qui réalise les fonctions suivantes :

— La méthode `execute()` implique l'instantiation d'un `Thread` depuis l'objet `Runnable` passé.
— Chaque `Thread` peut être mémorisé pour des futures traitements. Pour mémoriser les `Threads`, utiliser un conteneur `vector`.
— Une fonction `boolean isActive()` qui retourne vrai, s'il y a encore un `Thread` actif parmi les créés.
— Une fonction `void joinAll()` qui permet d'attendre la fin de l'exécution de tous les `Threads` du gestionnaire.

Implémentez et testez ce gestionnaire avec une classe qui implémente `Runnable` à votre choix.

R.

```
import java.util.concurrent.Executor;
import java.util.Vector;

public class ExecutorTest implements Executor {

    private  Vector T;
```

1. Pour utiliser les classes `Executor` et `ExecutorService` :
`import java.util.concurrent.Executors;`
`import java.util.concurrent.ExecutorService;`

```
    // constructeur
    ExecutorTest () {
        T=new Vector();
    }

     public void execute(Runnable r) {
         Thread th = new Thread(r);
   T.addElement(th);
         th.start();
     }

     public void joinAll() {
      int s = T.size();
      try {
       for (int n = 0; n<s; n++) {
        ((Thread)T.elementAt(n)).join();
        }
      } catch (InterruptedException e) {     }
     }

     public boolean isActive() {
     boolean res = false;
     int s = T.size();
      for (int n = 0; n<s; n++) {
       if ( ((Thread)T.elementAt(n)).isAlive()) res = true;
        }
     return res;
     }

     public static void main(String args[]) {
      ExecutorTest E = new ExecutorTest();

      for (int n = 0; n<5; n++) {
       TR R3=new TR(n);
       E.execute(R3);
      }
    System.out.println("mainÂă: les threads sont lancees" );
    System.out.println("Activites" + E.isActive() );

    E.joinAll();
    System.out.println("mainÂă: fin" );
    }
}
```

Exercice 2. *Utilisation simple des gestionnaires `Executor` retournés par `Executors`*

Rappelons que la classe `Executors` est un fabrique qui peut retourner différents instances de `Executor`. Nous allons reprendre la classe `TR` qui implémente `Runnable`.

1. Pour exécuter une unique tâche à la fois, on peut utiliser la classe suivante :

```
import java.util.concurrent.Executors;
import java.util.concurrent.Executor;

public class Application {
  public static void main(String [] args)
  { Executor es;
    TR t;
    System.out.println(" debut tache principale ");
    es = Executors.newSingleThreadExecutor();
    t = new TR(1,100,10);
    es.execute(t);
    t = new TR(2,150,10);
    es.execute(t);
    t = new TR(3,100,10);
    es.execute(t);
    System.out.println(" fin tache principale ");
  }
}
```

Remarque : cette classe crée trois instances `Runnable`.
Créez et gérez les tâches comme suggéré. Testez son fonctionnement.

2. Pourquoi la méthode `main` ne s'arrête pas ? Modifiez la pour terminer son exécution.

R. 1) le gestionnaire n'est pas stoppé.
2) Pour l'arreter : il faut faire un shutdown()...
3) Pour faire shutdown(), il faut utilisez l'interface ExecutorService...

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class Application {
  public static void main(String [] args)
  { ExecutorService es;
    TR t;
    System.out.println(" debut tache principale ");
    es = Executors.newSingleThreadExecutor();
    t = new TR(1,100,10);
    es.execute(t);
    t = new TR(2,150,10);
    es.execute(t);
    t = new TR(3,100,10);
    es.execute(t);
    es.shutdown();
    System.out.println(" fin tache principale ");
  }
}
```

3. Utilisez un gestionnaire similaire (pour une seule tâche exécutée) obtenu depuis la méthode `newFixedThreadPool(...)`

R.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class Application {
  public static void main(String [] args)
```

```
  { ExecutorService es;
    TR t;
    System.out.println(" debut tache principale ");
    es = Executors.newFixedThreadPool(1);
    t = new TR(1,100,10);
    es.execute(t);
    t = new TR(2,150,10);
    es.execute(t);
    t = new TR(3,100,10);
    es.execute(t);
    es.shutdown();
    System.out.println(" fin tache principale ");
  }
}
```

4. Modifiez la définition de l'`ExecutorService` dans `Application` pour autoriser l'exécution de deux tâches simultanément.Observez le résultat.

R.

```
    es = Executors.newFixedThreadPool(2);
```

Les deux tâches s'exécutent en paralléle.

5. Modifiez maintenant la méthode `main()` pour exécuter les trois tâches simultanément (attention á l'`ExecutorService`). Relancez l'application plusieurs fois pour voir si les affichages différent entre les différentes exécutions.

R.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class Application {
  public static void main(String [] args)
  { ExecutorService es;
    TR t;
    System.out.println(" debut tache principale ");
    es = Executors.newFixedThreadPool(3);
    t = new TR(1,100,10);
    es.execute(t);
    t = new TR(2,150,10);
    es.execute(t);
    t = new TR(3,100,10);
    es.execute(t);
    es.shutdown();
    System.out.println(" fin tache principale ");
  }
}
```

Les trois tâches sont exécutées en paralléle (les alternances entre les trois tâches ne sont pas parfaitement réguliéres).

6. Que peut-on dire sur la terminaison de l'application si l'on enlève l'appel de `shutdown()` ? Peut-on le remplacer par `shutdownNow()` ? Etudiez la documentation ;

R. Non, l'application continue de fonctionner car l'`ExecutorService` n'est pas arrêté.

7. Recherchez sur Internet la documentation correspondant aux fonctions `shutdown()` et `shutdownNow()` de la classe `ExecutorService`, et essayez les deux pour que l'application se termine.

R. Extrait de la documentation *Java* sur le site `http://docs.oracle.com/` :

An `ExecutorService` can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an `ExecutorService`. The `shutdown()` method will allow previously submitted tasks to execute before terminating, while the `shutdownNow()` method prevents waiting tasks from starting and attempts to stop currently executing tasks. Upon termination, an executor has no tasks actively executing, no tasks awaiting execution, and no new tasks can be submitted. An unused `ExecutorService` should be shut down to allow reclamation of its resources.

On peut donc ajouter `es.shutdown()` (ou `es.shutdownNow()`) á la fin de la méthode `main` pour arrêter l'`ExecutorService`. Attention, dans certains cas `es.shutdownNow()` arrête l'exécution et lance une exception...

Un commentaire : " shutdown() will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run

shutdownNow() will do the same AND will try to cancel the already submitted tasks by interrupting the relevant threads. Note that if your tasks ignore the interruption, shutdownNow will behave exactly the same way as shutdown. "

8. Les traces de l'exécution sont encore plus claires quand la pause `sleep` dans les tâches est aléatoire. Transformez votre classe `Activite` qu'elle fasse une pause aléatoire dans la méthode `faire()` chaque fois quand `sleep()` est appelée. Cherchez la documentation de la classe `Random`. Essayez vos classes en tournant trois tâches parallèlement.

R.

```
  import java.util.Random;

public class Activite {
    private int id;     // un identifiant
    private int delai;
    private int duree;
    private static char[] S1 = {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
        ' ',' ',' ',' ',' ',' ',' ',' ',' '};
    private  String B; // pour affichage
    // constructeur
    Activite (int nb, int pause, int lng) {
        id=nb;
  delai = pause;
  duree = lng;
  B = new String(S1,0,id); // la longueur de B est proportionnelle a id
    }
    public void faire() {
        Random rand = new Random();
  int attent;
  double f = 1.3333;  // une valeur arbitraire
        for (int nombre=1; nombre <duree; nombre++) {
    System.out.println("T" + id + " \t|" + B + 'x'); // on laisse une trace
      for (int i=1; i <10000000; i++) f *= 1.000001;  // des multiplications
          pour consommer le temps
      attent = rand.nextInt(199) + 1;
            try {
                Thread.sleep(pause); // milliseconds
            } catch (InterruptedException e) {  }
        }
  System.out.println("Fin activite T" + id);
    }
}
```

Exercice 3. *Interface* `Callable`

Comme nous avons vu, la méthode `void run()` depuis un `Runnable` ne permet pas de retourner une valeur (ni de lancer une exception).

L'interface `Callable<V>` corrige ce problème puis que sa méthode `call()` peut retourner une valeur de type `V`.

Les éléments de base de l'utilisation de `Callable<V>` sont :

— Créer la classe qui implément `Callable<V>` (qui définie la méthode `call()` retournant u objet de type `V`)
— Pour lancer l'exécution, on utilise la méthode `Future<V> submit(Callable<V> )` d'un `ExecutorService`.
— L'objet retourné est une instance de `Future<V>`
— La méthode `get()` de l'objet du type `Future<V>` permet de récupérer la valeur retournée par `call()`

1. Etudiez la documentation de la classe `Future<V>`.

2. Pour tester cette solution, créez une classe `TR` de l'exercice précédant : maintenant, elle doit implémenter `Callable<Integer>` et sa méthode `Integer call()` doit afficher des valeurs successives de 1 jusqu'à une valeur aléatoire entre 25 et 35 De plus, `call()` doit retourner la dernière valeur affichée. Le programme principale doit lancer 3 tâches, récupérer et afficher les valeurs retournées.

R.

```
// Fichier: TC.java
// ExempleÂă: Callable
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TC implements Callable<Integer> {
    private int nbr;
    // constructeur
    TC (int nb) {
        this.nbr=nb;
    }

    public Integer call() {
        Random r = new Random();
  int b = r.nextInt(11) + 25;
        for (int nombre=1; nombre <=b; nombre++) {
            System.out.println("Je suis F" + nbr + " " + nombre);
            try {
                Thread.sleep(200); // milliseconds
            } catch (InterruptedException e) { }
        }
        return b;
    }

    public static void main(String args[]) {
        TC e1 = new TC(1);
        TC e2 = new TC(2);
        TC e3 = new TC(3);
        int resultat;
        ExecutorService ex = Executors.newFixedThreadPool(3);
        Future<Integer> F1 = ex.submit(e1);
  System.out.println("mainÂă: F1 est lancee" );
```

```
        Future<Integer> F2 = ex.submit(e2);
  System.out.println("mainÂă: F2 est lancee" );
        Future<Integer> F3 = ex.submit(e3);
  System.out.println("mainÂă: F3 est lancee" );
        while (!F1.isDone() || !F2.isDone() ||!F3.isDone() ) {
              System.out.println("attente ");
            try {
                Thread.sleep(500); // milliseconds
            } catch (InterruptedException e) {   }
         }
         try {
           resultat = F1.get();
           System.out.println("F1 : " + F1.get() + " F2 :  " + F2.get() +
             " F3 :  " + F3.get());
           } catch (ExecutionException e) {  }
             catch (InterruptedException e) {  }
        ex.shutdown();
  System.out.println("mainÂă: termine" );
    }
} // fin TC
```