Processus objets – Java threads

Table des matières

I. Qu'est-ce qu'un thread Objet ?

C'est un objet qui correspond à un processus indépendant ! Ainsi, on peut créér un thread. Il faut ensuite le lancer, pour qu'il s'exécute. C'est ici que l'on a une différence avec le `fork()` de C, par exemple. Car fork crée *et* lance l'exécution. Avec un objet, il est possible de différencier ces deux actions.

II. Qu'est-ce qu'un thread en Java ?

A) Une classe héritant de la classe Thread

Cette classe doit implémenter une méthode `run()`. Cette méthode est appelée de manière un peu particulière puisqu'il faut faire un appel `start()` pour la lancer !

```
public class Exemple {
   public static void main (String[] args) {
      Thread t = new monThread();
      t.start();
   }
}

class monThread extends Thread {
   public void run() {
      for (int i = 1; i <= 1000; i++) System.out.println(i);
   }
}
```

B) Une classe implémentant l'interface runnable

En effet, comme l'héritage multiple est impossible en Java, cela se fait par une interface. ATTENTION, pour exécuter un objet implémentant runnable, il faut créér un thread à l'aide de l'objet runnable.

```
public class MyRun {
   public static void main (String[] args) {
      Thread t = new Thread( new monRunnable() );
      t.start();
   }
}

class monRunnable implements Runnable {
   public void run() {
      for (int i = 1; i <= 1000; i++) System.out.println(i);
   }
}
```

```
public class monThreadBis extends Thread {
   public void run() {
      for (int i = 1; i <= 10; i++) {
         System.out.println(i + "  " + getName());
      }
   }

   public static void main (String[] args) {
      new monThreadBis().start();
      new monThreadBis().start();
   }
}
```

III. Construction d'un thread

A) getname() et setname(String)

Par défaut le nom d'un thread est "Thread-##".

B) toString()

Nom, priorité, groupe ... du thread qui lance cette méthode.

```
public class Nom extends Thread {
   public void run() {
         System.out.println( toString() );
   }

   public static void main (String[] args) {
      new Nom().start();
      new Nom().start();
   }
}
```

C) Gestion de la priorité : getpriority() et setpriority(int)

Ces deux méthodes permettent donner des priorités aux threads, sachant que celle-ci peut varier entre `MAX_PRIORITY` et `MIN_PRIORITY`, et qu'elle a une valeur par défaut de `NORM_PRIORITY`

```
import java.io.*;

public class MonThread6 extends Thread {
   static Th1 th1;
   static Th2 th2;
   public static void main (String[] args) {
      th1 = new Th1();
      th2 = new Th2();
      th1.start();
      th2.start();
   }
}

class Th1 extends Thread {
   private boolean arret = false;
   private boolean interruption = false;

   public void run() {
      setPriority(MAX_PRIORITY);
```

```
      int i = 0;
      while (!arret){
         System.out.println(i++);
      }
   }

   public void arrete() {
      arret = true;
   }
}

class Th2 extends Thread {
   public void run() {
      setPriority(MIN_PRIORITY);
      long duree = 10000;
      long t1 = System.currentTimeMillis();
      long t = 0;
      while (t < duree) {
         t = System.currentTimeMillis() - t1;
      }
      MonThread6.th1.arrete();
      System.out.println(t);
   }
}
```

IV. Contrôle d'un thread Java

A) yield()

Interrompt le déroulement de façon quasi imperceptible afin de laisser du temps pour l'exécution des autres threads d'un même niveau de priorité.

B) interrupt()

Interrompt le thread.

C) sleep()

Interrompt le déroulement pendant une durée spécifiée en millisecondes (+ éventuellement en nanosecondes).

```
public class monThread3 extends Thread {
   public void run() {
      for (int i = 1; i <= 10; i++) {
         System.out.println(i + "  " + getName());
         try { sleep(0,120); } catch (InterruptedException e) {}
      }
   }

   public static void main (String[] args) {
      new monThread3().start();
      new monThread3().start();
   }
}
```

D) join()

Attend la fin de l'exécution (lance une interruption si le thread est interrompu).
`join(long millis)` (resp. `join(long m, int n)`) attend la fin de l'exécution du thread au plus pendant m millisecondes (resp. et n nanosecondes).

V. Groupes de threads

Il est possible de regrouper des threads dans un groupe.

A) activeCount()

Nombre de threads dans le groupe du thread qui lance cette méthode.

B) getThreadGroup()

Renvoie le groupe auquel appartient le thread qui lance cette méthode.

C) enumerate(Thread [] tabl)

Tableau de `handle` sur les threads de ce groupe et de ses sousgroupes.

```
import java.io.*;

public class UnGroupe extends Thread {
   public static void main (String[] args) {
      ThreadGroup tg = new ThreadGroup("Groupe");
      new Th1(tg).start();
      new Th2(tg).start();
   }
}

class Th1 extends Thread {
   private boolean arret = false;
   Th1(ThreadGroup tg) {
      super(tg, "Th1");
   }

   int i = 0;

   public void run() {
      while (!arret){
            System.out.println(++i);
      }
   }

   public void arrete() {
      arret = true;
      System.out.println("Boucle arretee a " + i);
   }
```

```
}

class Th2 extends Thread {
   Thread[] ta = new Thread[2];

   Th2(ThreadGroup tg) {
      super(tg, "Th2");
   }

   public void run() {
      enumerate(ta);
      String s1 = "";
      BufferedReader r =    new BufferedReader(new InputStreamReader(System.in));
      while (!s1.equals("s")) {
         try {
              while (s1 == "") {
                  s1 = r.readLine();
            }
            if (!s1.equals("s")) {
               s1 = "";
            }
         }
           catch (IOException e) {
         }
      }
      ((Th1)ta[0]).arrete();
   }
}
```

VI. Synchronisation

A) Prise de verrou en exclusion mutuelle

Lorsque deux processus doivent accéder à un même objet, il peut être nécessaire qu'ils ne le fassent qu'un seul à la fois. Pour réaliser cela, il est possible de déclarer une méthode ou un bloc de code comme étant "synchronisé". L'appel à `synchronized(variable)` pose un "verrou" sur `variable`. Ainsi, la portion `code1` entre `synchronized(variable){ code1 ; }` n'est exécutée que lorsque le verrou de la variable est libre (i.e. si aucun autre processus n'a posé de verrou sur cette variable).

```
class SynchronizeBloc {
    public static void main(String args[]) {
        Shared shared = new Shared();
        CustomThread thread1 = new CustomThread(shared);
        CustomThread thread2 = new CustomThread(shared);
        CustomThread thread3 = new CustomThread(shared);
        CustomThread thread4 = new CustomThread(shared);
    }
}

class CustomThread extends Thread {
    Shared shared;

    public CustomThread(Shared shared) {
        super();
        this.shared = shared;
        start();
    }

    public void run() {
        System.out.println("Debut de run de " + getName());
        synchronized(shared) {
            System.out.println(" Debut de Bloc de " + getName());
            shared.doWork(Thread.currentThread().getName());
            System.out.println(" Fin du Bloc de " + getName());
        }
        System.out.println("Fin de run de " + getName());
    }
}

class Shared {
    void doWork(String string) {
        System.out.println(" Demarrage de la partie partagee de " + string);
```

```
        try { Thread.sleep((long) (Math.random() * 500));
        } catch (InterruptedException e) {}

        System.out.println(" Fin de la partie partagee de " + string);
    }
}
```

Il est en outre possible de synchroniser la totalité d'une méthode d'une classe, et non une variable. Il suffit de déclarer cette methode comme `synchronized`. Alors un seul thread à la fois peut lancer cette méthode.

B) Attente : wait, notify, notifyAll

Les threads ont souvent besoin de se coordonner mutuellement, en particulier quand le résultat de l'un est utilisé par un autre. Une manièrfe de se coordonner est d'utiliser les méthodes suivantes :

`wait()` Met un processus en sommeil jusqu'à un appel à notify ou à notifyAll sur le même objet. Cette méthode peut aussi prendre en argument en temps d'attente maximal en millisecondes (et en nanosecondes si il y a un deuxième argument).

`notify()` Redémarre le premier processus qui a appelé wait sur le même objet.

`notifyAll()` Redémarre tous les threads qui ont appelé wait sur le même objet.

```
class Shared
{
    int data = 0;

    synchronized void doWork() {
        try { Thread.sleep(1000);
        } catch(InterruptedException e) {}

        data = 1;
        notify();
    }

    synchronized int getResult() {
        try {
            wait();
        } catch(InterruptedException e) {}

        return data;
    }
}

class CustomThread1 extends Thread {
    Shared shared;
```

```
    public CustomThread1 (Shared shared) {
        super(); this.shared = shared; start();
    }

    public void run() {
        System.out.println("Le resultat est " + shared.getResult());
    }
}

class CustomThread2 extends Thread {
    Shared shared;

    public CustomThread2 (Shared shared) {
        super(); this.shared = shared; start();
    }

    public void run() {
        shared.doWork();
    }
}


class WaitNotify {
    public static void main(String args[]) {
       Shared shared = new Shared();
        CustomThread1 thread1 = new CustomThread1(shared);
        CustomThread2 thread2 = new CustomThread2(shared);
    }
}
```

VII. Démons

Un démon est un thread qui reste en mémoire pour servir d'autres processus. Il ne se termine pas de lui-même, il est automatiquement terminé s'il ne reste plus que des démons à s'exécuter. Un thread devient un démon par l'appel à la méthode `setDaemon(true)`. Cet appel ne peut se faire qu'*avant* l'appel à `start()`.

```
import java.awt.Toolkit;

public class Message extends Thread {
    public static void main (String[] args) {
        ThreadGroup email = new ThreadGroup("email");
        GotMail mozilla = new GotMail(email);
        ServeurMail S = new ServeurMail(email);
        mozilla.start();
}   }

class GotMail extends Thread {
    boolean MessageEnAttente;
    GotMail(ThreadGroup tg) { super(tg, "GotMail");
        MessageEnAttente = false;
        setDaemon(true); setPriority(MIN_PRIORITY);
    }
    public void run() {
        while(true) { if (MessageEnAttente) avertir(); }
    }
    public void avertir() {
        Toolkit.getDefaultToolkit().beep();
        MessageEnAttente = false;
    }
    public void newmail() { MessageEnAttente = true; }
}

class ServeurMail extends Thread {
    double Mail;
    ServeurMail(ThreadGroup tg) { super(tg, "Serveur"); start(); }
    public void run() {
        Thread [] ta = new Thread[2]; enumerate(ta);
        for(int i = 0; i < 10; ++i) {
            try {  sleep((long) (Math.random() * 10000)); }
            catch (InterruptedException e) {}
            ((GotMail)ta[0]).newmail();
            System.out.println("Message " + i);
        }
}   }
```