Threads : les bases

Introduction

Tous les utilisateurs d'ordinateurs, qu'ils soient programmeurs, *informaticiens experts*, ou simples utilisateurs, s'accordent pour dire qu'il est insupportable de travailler sur une machine qui « rame » ou avec des programmes si lents d'exécution qu'il est facile de croire que le système a « planté ». La technique des *Threads* sert notamment à améliorer les performances d'un programme. Bien sûr, rien ne remplace une configuration « musclée » et les Threads servent surtout à donner à l'utilisateur une illusion de rapidité (heureusement ils ne se limitent pas à cela). En effet, imaginez un programme qui, une fois lancé, fait des calculs mathématiques qui doivent durer 2 heures. L'utilisateur ne pourra plus arrêter l'opération en cours et devra patienter 2 heures avant de pouvoir utiliser son ordinateur. Cette situation est vraiment embêtante, c'est pourquoi la plupart des logiciels qui exécutent de longues opérations ont une commande pour annuler l'opération en cours : habituellement la touche ESC (Echap) ou un bouton « Annuler ». Mais un problème se pose : un programme Java ne peut pas faire 2 choses à la fois : faire ses calculs mathématiques et traiter les clics de souris sur ce bouton « Annuler ». Il faudra donc attendre que toutes les opérations en cours soient terminées pour que le bouton « Annuler » réagisse aux actions de l'utilisateur. Nous sommes revenus à notre point de départ. C'est ici qu'interviennent les Threads : ils permettent d'exécuter plusieurs actions simultanées. Tout programme Java possède au moins 1 thread : le thread principal. Dans notre exemple, le thread principal servira aux opérations mathématiques et un thread « secondaire » sera chargé de « capter » les clics de la souris sur le bouton « Annuler » et de réagir en fonction.
Ce chapitre est une introduction aux threads qui sont souvent un point difficile d'un langage de programmation. Beaucoup de programmeurs venant du C/C++ ont été surpris de voir juxtaposés « Java simple » et « Java multi-hreads ». Mais c'est vrai, Java permet d'utiliser assez simplement les threads (pour des opérations de base).
Nous allons voir dans un exemple concret comment utiliser les threads.

Créer un nouveau thread

On crée en Java un objet thread tout comme les objets « normaux ». Il existe plusieurs manières de créer et d'activer un nouveau thread ; nous allons voir dans notre exemple l'*implémentation* des méthodes de l'interface **Runnable**.
Par défaut, une classe qui travaille avec l'interface Runnable doir implémenter les méthodes suivantes :

- init() = créer un nouveau thread, l'activer
- run() = ce que le thread doit faire durant son exécution
- stop() = comment arrêter ce thread (méthode optionelle)

Exemple

L'implémentation des méthodes start() et stop() (jamais utilisée) dans notre exemple est basique et peut être réutilisée dans vos programmes (tant que le mécanisme est simple).
Nous allons créer une horloge. Nous devons utiliser une thread en imaginant que l'utilisateur ne pourrait rien faire tant que le chronomètre ne se serait pas arrêté si ce thread n'était pas utilisé.
Structure du programme :

- interface graphique simple grâce à Swing (1 cadre, 1 label).
- Boucle **for** infinie avec une variable **i** qui indique le nombre de secondes écoulées.
- Un thread qui « sommeille » pendant 1000 ms (ou 1 seconde) avant de permettre à la boucle de faire un nouveau passage. (grâce à la méthode java.lang.Thread.sleep(int i))

Note : pour l'affichage on utilise une instruction de la forme : **label.setText(label.getText() + mon_texte) ;** de manière à ce que le texte ne soit pas effacé à chaque passage de la boucle.
Note2 : l'opérateur modulo (%) (c'est le signe « pourcentage » sur le clavier) permet d'obtenir le *reste* d'une division. Ainsi, 62/60 (i/60 pour i=62) donnera 1. Le reste de la division **i/60** est toujours tels que 0<reste<60 pour tout réel positif i. Donc, pour i > 60, i%60 donnera le nombre de secondes et 1%3600 pour i > 3600 le nombre de minutes.

```
import java.awt.*;
import com.borland.jbcl.layout.*;
import java.util.*;
import javax.swing.*;

public class Chrono extends JFrame implements Runnable {
  XYLayout xYLayout1 = new XYLayout();
  Date date = new Date();
  JLabel jLabel1 = new JLabel();

  // C'est un objet THREAD (initialisé dans la méthode start()).
  Thread thread;

  public Chrono() {

    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  // méthode par défaut de JBuilder pour l'interface graphique
  private void jbInit() throws Exception {
    jLabel1.setText("Lancé depuis: ");
    this.getContentPane().setLayout(xYLayout1);
    xYLayout1.setWidth(415);
    xYLayout1.setHeight(41);
    this.setSize(400, 70);
    this.setResizable(false);
    this.setVisible(true);
    this.getContentPane().setBackground(Color.lightGray);
    this.show();
    start();
    run();
    this.getContentPane().add(jLabel1, new XYConstraints(8, 8, 402, 23));
  }


 // méthode START du thread: créer un nouveau thrad et le lancer
 public void start() {
  if (thread == null) {
    thread = new Thread(this);
  }
  thread.start();
 }

 public void stop() {
  if (thread != null)
    thread = null;
 }

  // ce que le thread doit faire durant son exécution:
 public void run() {
  /* on peut avoir plusieurs threads dans une même classe et 1 seule
     méthode run(), on doit donc savoir si le thread qui nous concerne
     est actif (si il est le thread "courant"). */
   while(Thread.currentThread() == thread) {
    // les threads sont susceptibles de générer des exceptions
     try {
       /*Vous remarquerez que le boucle for ne spécifie pas de valeur
       à tester pour savoir si elle doit continuer ou s'arrêter: sans
       valeur spécifique, la boucle ne s'arrête jamais */
       for(int i = 0;; i++) {
         // rafraîchir l'écran
       repaint();
         // attendre 1 seconde ou 1000 ms
       Thread.sleep(1000);
         // tanta que i est inférieur à 60, on ne s'occupe que des secondes
       if (i < 60) {
```

```
        jLabel1.setText("Lancé depuis: " + i +" secondes");
        }
          // quand i est entre 1 et 2 minutes, le mot "minute doir être au singulier"
        else if (i > 60 && i < 120) {
        jLabel1.setText("Lancé depuis: " + i/60 + " minute, soit " + i +" secondes");
        }
         // ensuite, on s'occupe des minutes normalement
        else if (i > 120 && i < 3600) {
        jLabel1.setText("Lancé depuis: " + i/60 + " minutes " + i%60 +" secondes");
        }
          // "heure" au singulier entre 1 et 2 heures
        else if (i > 3600 && i < 7200) {
        jLabel1.setText("Lancé depuis: " + i/3600 + " heure " + i%60 + " minutes,
soit " + i%3600 +" secondes");
        }
          // ensuite "heures" au pluriel et en "triatement normal"
        else if (i > 7200) {
        jLabel1.setText("Lancé depuis: " + i/3600 + " heures " + %/60 + " minutes,
soit " + i%3600 +" secondes");
        }
        }
       }catch(Exception err) {}
     }
    }
    public static void main(String[] args) {
      // lancer le constructeur de la classe
       new Chrono();
    }
   }
```

Notre chronomètre n'est pas très précis : le thread s'arrête pendant 1 seconde pas le traitement de toutes ces instructions **if**, même si elles ne prennent que quelques millièmes de seconde retardent le programme. Si vous ne modifiez pas le code et que vous laissez tourner le programme pendant plusieurs jours, vous constaterez qu'il « retarde » légèrement. Pour résoudre ce problème, on doit définir une valeur comme 995ms de sommeil pour notre thread, mais comment savoir exactement le temps que prendra sur une machine données le traitement des ces boucles ? La question reste sans réponse.