Threads, les bases

Michel CASABIANCA
casa@sweetohm.net

Cet article est une introduction aux threads en Java.

Pourquoi les threads ?

L'idée des threads [^1] est de partager le temps CPU entre plusieurs tâches que le programme effectue simultanément. Je vois deux cas où l'emploi des threads est utile :

1. Pour des raisons de performances : des programmes comportant une interface graphique passent l'essentiel de leur temps à attendre des ordres de l'utilisateur (entrées clavier ou souris). Il est alors intéressant d'utiliser ces temps morts pour effectuer d'autres taches utiles.
2. Certains programmes doivent faire tourner plusieurs tâches simultanément. C'est le cas d'un navigateur : il doit gérer les entrées des utilisateurs et charger des fichiers (textes, images, sons...) simultanément.

Pour toutes ces raisons, il est quasiment indispensable à tout programmeur Java de maîtriser l'emploi des threads. La création de threads est de plus très simple en Java, alors pourquoi s'en priver ? Lorsqu'on a apprivoisé les threds, on ne peut plus s'en passer, et on ne conçoit plus un programme comme on en avait l'habitude (suite linéaire d'instructions), on pense threads...

Comment ?

Il y a deux possibilités pour créer un thread en Java :

1. Créer un objet qui hérite de la classe Thread.
2. Créer un objet qui implémente l'interface Runnable.

Dans les deux cas, la tâche effectuée par le thread est implémentée dans la méthode run(). Si l'on crée un objet qui hérite de la classe Thread, on doit donc surcharger cette méthode. Si on crée un objet qui implémente l'interface Runnable, on doit y définir la méthode run(). On doit aussi y instancier un objet de type Thread (que l'on peut appeler thread par exemple). On lancera le thread par thread.start() et on l'arrètera par l'appel thread.stop() [^2]. L'interface Runnable ne comporte qu'une méthode, c'est précisément cette méthode run().

Plutôt qu'un long discours, un petit exemple me semble plus parlant. Pour illustrer cet article, je me propose de développer une applet d'affichage d'images sous forme de diaporama. Je présente ci-dessous un premier exemple sans thread et nous verrons comment on peut améliorer cette applet en utilisant un thread.

Sans thread

Cette applet affiche des images les unes à la suite des autres. Pour passer d'une image à la suivante, l'utilisateur clique sur le bouton [>], ce qui appelle la méthode afficherImage() dont voici le source :

```
/* affichage d'une image */
void afficherImage() {
  /* on affiche le numéro de l'image dans le compteur */
  compteur.setText(index+1+" / "+nom.length);
  /* si l'image n'est pas chargée, on la charge */
  if(image[index]==null)
    image[index]=getImage(getDocumentBase(),nom[index]);
  /* on affiche l'image demandée */
  Graphics g=ecran.getGraphics();
  Rectangle r=ecran.bounds();
  g.drawImage(image[index],(r.width-image[index].getWidth(this))/2,
    (r.height-image[index].getHeight(this))/2,this);
}
```

Cependant, il est perdu énormément de temps dans cette applet : pendant que l'utilisateur regarde une image, l'applet ne fait rien. On peut utiliser ces temps morts pour charger les autres images. C'est que nous allons voir dans l'exemple suivant.

Avec Thread

Notre applet modifiée doit dorénavant répondre aux ordres de l'utilisateur (premier thread) et charger les images suivantes en tâche de fond (deuxième thread). Je vais utiliser la deuxième méthode (objet implémentant la classe Runnable). On doit alors modifier le code comme suit :

Runnable

Il faut que la classe de l'applet soit déclarée implements Runnable. Pour ce faire, la classe sera déclarée comme suit :

```
public class AvecThread extends java.applet.Applet implements Runnable
```

Méthode run()

Il faut fournir au programme le code à exécuter dans le thread au sein de la méthode run() :

```
/** charge les images une par une */
public void run() {
  for(int i=0;i<nom.length;i++) {
    /* si l'image est déjà chargée, on passe à la suivante */
    if(image[i]!=null) continue;
    /* on charge l'image */
    image[i]=getImage(getDocumentBase(),nom[i]);
    /* on attend la fin de son chargement */
```

```
        MediaTracker mt=new MediaTracker(this);
        mt.addImage(image[i],0);
        try {mt.waitForID(0);}
        catch(InterruptedException e) {}
      }
    }
```

Méthode start()

Il faut enfin lancer le thread au démarrage de l'applet. On doit donc transformer la méthode start() de l'applet :

```
    /** affiche l'image N°1, et on lance le chargement des suivantes */
    public void start() {
      afficherImage();
      tache=new Thread(this);
      tache.start();
    }
```

Le programme se comporte comme la première version, mais il lance le threa au démarrage (la méthode start() est appelée par la machine virtuelle Java au démarrage de l'applet). Le coeur de ce thread est la méthode run() qui s'exécute en parallèle du thread principal. Cette méthode run() charge les images une à une (il attend leur chargement avec un MediaTracker, dont l'utilisation est détaillée dans l'article "Attendre le chargement des images" sur ce site).

On peut encore se poser la question suivante : comment le thread sait-il qu'il doit faire tourner cette méthode run() ? Tout simplement par ce qu'on lui a passé en argument de son constructeur un objet Runnable. Cet objet est this en l'occurence, donc l'applet elle même. Notre applet implémentant l'interface Runnable, le thread "sait" qu'il peut appeler la méthode run() de notre applet, ce qu'il fait lorsqu'on appelle start(). Autre remarque en passant : la méthode start() de l'applet, appelée au démarrage, et la méthode start() d'un thread, n'ont rien à voir.

Pour terminer, il faudrait parler de l'autre méthode qui consiste à créer un objet qui hérite de Thread. Je ne la traiterai pas en détail, son implémentation étant très semblable à celle vue ci-dessus. Il suffit d'implémenter une classe ChargeurImage par exemple qui hérite de Thread et qu comporte une unique méthode run() identique à celle vue ci-dessus. Dans le code de l'applet (sans méthode run()), on créera une instance de cette classe et on appellera sa méthode start() dans la méthode start() de l'applet. Cependant, il faudra définir les variables de l'applet comme static pour pouvoir y accéder du Thread.

Conclusion

Attention, les threads sont une drogue dure ! Quand on y a goûté... J'ai vu des programmeurs gravement intoxiqués qui en voyaient partout (des roses avec une trompe ?), et en saupoudraient leur code à tord et à travers. On ne peut pas TOUT résoudre avec un thread.

Pour finir, je ne peux que vous recommander de vous pencher sur la classe Thread qui comporte de nombreuses méthodes dont je n'ai pas parlé ici. Cet article n'a fait qu'effleurer le sujet (très vaste)

des threads (des bouquins entiers lui sont consacrés !). En particulier, l'utilisation de threads pose des problèmes de synchronisation : deux threads peuvent intéragir s'ils modifient simultanément les mêmes variables, qu'il peut être nécessaire de protéger.

Les threads avancés seront peut être le sujet d'un prochain articles si j'en trouve le loisir (je ne suis moi même que faiblement multitache :) [^3].

On trouvera les sources des exemples de cet article en suivant ce lien.

[^1]: Le terme processus n'est pas adapté pour traduire la notion de thread car un thread ne donne pas lieu à un processus système.

[^2]: Il est en fait déconseillé d'arrêter ainsi un thread, mais ceci est une autre histoire !

[^3]: Cette remarque date tout de même de 1997, je ne suis vraiment pas multithreadé :o)