IFT1025 automne 2009

Miklós Csűrös

27 octobre 2009

Procéssus légers

Threads

Un **thread** est un *fil d'exécution* ou *procéssus léger*. Les threads partagent les mêmes ressources (en particulier, un espace mémoire en commun). En Java : `java.lang.Thread`.
Constructeurs :

- **Thread**() constructeur vide
- **Thread**(`Runnable` *r*) constructeur avec tâche *r*

Exécution dans le fil : `void` **start**() → exécution de la méthode **run**() de `Thread` qui appelle `void` **run**() de l'interface `Runnable`.
Définition d'une tâche :

Implantation de l'interface `Runnable` et sont utilisation lors de l'instanciation du `Thread` — flexible (on peut implanter l'interface par n'importe quelle classe), mais plus de problèmes do cohérence (plusieurs `Thread`s travaillent avec le même objet.

Héritage de la classe `Thread` : extension de la classe et redéfinition de sa méthode **run**() — moins général

Suspension d'exécution. Utiliser la méthode statique `Thread`.**sleep**(`int` millisecondes). `sleep` peut lancer des `InterruptedException` → capturer l'exception pour terminer l'exécution immédiatement.

Interruption. Utiliser *t*.**interrupt**() avec le `Thread` *t* dans un autre fil d'exécution. Si *t* est en attente (p.e., en **sleep**()), alors une exception `InterruptedException` est lancée. Si *t* est en exécution, alors *t* doit tester l'interruption explicitement par `Thread`.**interrupted**(). Statut d'interruption : faux au début, affecté à vrai par **interrupt**(). Capture d'un `InterruptedException` ou lecture statique `Thread`.**interrupted**() réinitialise le statut à faux. Par contre, la méthode **interrupted**() de l'objet ne change pas le statut.

```
public class MaClasse extends UneAutreClasse implements Runnable
    ... code quelconque pour la classe
    public void run()
    {
        try
        {
            while(condition){ ... sleep(x);} // boucle avec attente
        } catch (InterruptedException e) {return ;} // interruption
        boolean fini = false;
        while(condition)
        {
            ... calcul compliqué
            if (Thread.interrupted()){ fini=true; break;}
        }
        if (fini) return; // interruption demandée
        ... continuer calcul compliqué
    }
}
```

Exclusion mutuelle. Utiliser `synchronized` dans la signature de méthodes : un seul `Thread` peut exécuter une telle méthode à la fois. Cela évite des problèmes de cohérence de données lors d'exécution interlacée (plusieurs `Thread`s manipulent les mêmes variables).

```
    // sans synchronisation, plusieurs Threads
    // peuvent arriver à des points dénotés par * à même temps
    synchronized public void changeEtatUNO()
    {
        ... changer des valeurs de variables d'objet
        ... *
        ... affichage de résultats
    }
    synchronized public void changeEtatDOS()
    {
        ... changer des valeurs de variables d'objet
        ... *
        ... affichage de résultats
    }
```

Fonctionnement : il existe un verrou associé à chaque objet et des objets `Class` associés avec chaque classe. Pour exécuter du code `synchronized`, il faut acquérir le verrou, et un seul `Thread` peut le posséder à la fois. On peut isoler même portions du code par le syntaxe `synchronized`(v).

```
    ... code non-synchronisé
    synchronized(v) // avec verrou v
    {
        ... code exécuté en exclusion mutuelle
    }
    ... code non-synchronisé
```

Rendez-vous. Attendre qu'un `Thread` t soit terminé : t.**join**().

Coopération. Attente/notification : méthodes **wait**() et **notify** de chaque `Object`. Fonctionnement : utilise le moniteur associé avec l'objet : moniteur = verrou +

ensemble en attente. À `wait()`, le moniteur est liberé (donc on a du l'acquérir par `synchronized`) et le `Thread` se met dans l'attente. Plusieurs `Thread`s peuvent attendre à un événement associé au même moniteur : **notify** en choisit un pour réveiller "au hasard». Le moniteur n'est pas libéré par **notify**() : il est pris jusqu'à la fin du bloc `synchronized`. On peut réveiller tous les procéssus en ettente par **notifyAll**().

Dans thread 1 :

```
// code thread 1
synchronized(o)
{
    try
    {
        ...
        o.wait(); // attente avec moniteur o
    } catch (InterruptedException e)
    {
     // interruption arrivée pendant l'attente
    }
}
```

Dans thread 2 :

```
// code thread 2
synchronized(o){o.notify(); }
```

Exemple : producteur – consommateur

Swing

Ne pas exécuter du code compliqué dans le fil d'événéments ! Situation typique : événement capturé par un `Listener` → création d'un **fil de travail** pour performer du calcul ; affichage du **progrès de calcul** en parallèle → mise à jour du GUI avec le résultats du calcul quand il termine ; enlever l'affichage du progrès. On peut implanter ce comportement désiré par une construction explicite ou en se servant d'un `SwingWorker`. Pour afficher le progrès de calcul, on utilise les éléments graphiques de Swing `javax.swing.ProgressMonitor` (fenêtre) ou `javax.swing.ProgressBar` (bande de progrès).

Construction explicite.

1. création d'un fil de travail `Thread` t par l'implantation de l'interface `Runnable`. Progrès de calcul stocké par un objet p – mis à jour de temps en temps durant le calcul.
2. initialisation de l'affichage de progrès avec `ProgressMonitor` ou `ProgressBar`.
3. lancer un `Timer` avec `ActionListener` qui prend la valeur de p et met l'affichage de progrès à jour par **setProgress**(`int` v) de `ProgressMonitor` ou **setValue**(`int` v) de `JProgressBar`, et vérifie la terminaison du calcul.
4. exécution du fil de travail avec t.**start**()
5. quand le calcul se termine (aperçu par l'`ActionListener` du `Timer`) : arrêter le `Timer`, enlever l'affichage de progrès, et mettre à jour le GUI avec le résultat.

SwingWorker. Pour créer un fil de travail avec suivi de progrès, créer une sous-classe de `SwingWorker<T,V>`. Méthodes :

- ⋆ `protected T` **doInBackground**() : code à exécuter sur le fil de travail
- ⋆ `protected void` **setProgress**(`int` v) : génère `java.beans.PropertyChangeEvent` dans le fil d'exécution
- ⋆ **addPropertyChangeListener**(`PropertyChangeListener` P) : P doit implanter `void` **propertyChange**(`PropertyChangeEvent` e). Il faut examiner `String` e.`getPropertyName`() : pour des changements de progrès, on a la propriété `progress`.
- ⋆ `protected void` **publish**(`V...` chunks) : envoie des données de type `V` pour communication avec le fil d'événements
- ⋆ `protected void` **process**(`List<V>`) : reçu de données publiées, executé dans le fil d'événements (affichage de résultats partiels)
- ⋆ `protected void` **done**() : executé dans le fil d'événements après `doInBackgound()` finit
- ⋆ `void execute`() : lance l'exécution

```
private ProgressMonitor progressMonitor;
private UneTache tache;

class UneTache extends SwingWorker<Void,Void>
{
    public Void doInBackground()
    {
        setProgress(0);
        while (!isCancelled() && condition)
        {
            ... calcul compliqué
            setProgress(x); // va de 0 à 100 avec les itérations
        }
    }
    public void done()
    {
        progressMonitor.setProgress(0); // enlever le moniteur
    }
}
void faireTache()
{
    progressMonitor
        = new ProgressMonitor(panel, "Tâche longue", "", 0, 100);
    progressMonitor.setProgress(0);
    tache = new UneTache();
    tache.addPropertyChangeListener(this);
    tache.execute();
}
public void propertyChange(PropertyChangeEvent evt)
{
    if ("progress" == evt.getPropertyName() )
    {
        progressMonitor.setProgress(evt.getNewValue());
        if (progressMonitor.isCancelled())
            tache.cancel(true);
}
```