Université du Luxembourg 2006–2007

D.U.T. Informatique 2ème année Semaine 44

Programmation avancée – Java

TD n°5 : Programmation concurrente

Sebastien.Varrette@imag.fr

Le but de ce TD est de manipuler des threads et de gérer les problèmes de synchronisation entre eux. Comme tout processus, un thread (on dit aussi *processus léger*) existe principalement sous trois états : **en exécution** (running), **prêt** (en attente du processeur) ou **bloqué**. Pour créer une classe de processus léger (thread) `Toto`, on a deux possibilités :

1. `Toto` hérite de la classe Thread comme dans l'exemple suivant :

```
public class Toto extends Thread {
    public void run() {...}
};
Toto t = new Toto();
t.start ();
```

2. `Toto` implémente l'interface `Runnable` et est passé en argument au constructeur `Thread` pour son exécution. Exemple :

```
public class Toto implements Runnable {
    public void run() {...}
};
Toto t = new Toto();
new Thread(t).start();
```

Les threads intègrent les notions de priorité et de synchrinisation. Parmi les méthodes les plus intéressantes, on distinguera :

- `start` (place un thread dans l'état prêt) et `run` (contient la tâche à effectuer) ;
- `sleep(n)` permet de faire une pause de `n` ms ;
- `yield` permet à un autre thread de prendre la main ;
- `getPriority` et `setPriority` permettent de changer la priorité d'exécution (qui varie de 1 à 10) ;
- `isAlive` permet de tester si un thread est vivant (c'est à dire à été démarré par `start` et que sa méthode `run` n'est pas encore terminée. Le thread vivant est donc prêt, bloqué ou en cours d'exécution).

Un thread s'arrête quand il termine sa méthode `run`. Il doit mourir "naturellement" : on ne peut plus l'arrêter en lui adressant une méthode particulière (`stop` est deprecated). On préférera boucler sur un booléen dont on changera la valeur au besoin. Considérez par exemple les deux programmes suivants :

```
public class PrintWithoutThread {
    final static int MAX_PRINT = 5;
    private String _name;
```

```
    public PrintWithoutThread(String name) { _name = name; }
    String getName()              { return _name; }

    public void start() { // une méthode comme une autre
        for (int i=0; i<MAX_PRINT; i++)
            System.out.println("[" + getName() + "] Message nř" + i);
    }
    public static void main (String args[]) {
        PrintWithoutThread t1 = new PrintWithoutThread("Thread 1");
        PrintWithoutThread t2 = new PrintWithoutThread("Thread 2");
        t1.start ();
        t2.start ();
        for (int i=0; i<MAX_PRINT; i++)
            System.out.println("Main task nř" + i);
    }
};
```

```
public class PrintWithThread extends Thread {
    final static int MAX_PRINT = 5;
    PrintWithThread(String name) { super(name); }

    public void run() {
        for (int i=0; i<MAX_PRINT; i++) {
            System.out.println("[" + getName() + "] Message nř" + i);
            Thread.yield(); // on passe la main au processus suivant
        }
    }
    public static void main (String args[]) {
        PrintWithThread t1 = new PrintWithThread("Thread 1");
        PrintWithThread t2 = new PrintWithThread("Thread 2");
        t1.start ();
        t2.start ();
        for (int i=0; i<MAX_PRINT; i++) {
            System.out.println("Main task nř" + i);
            Thread.yield(); // on passe la main au processus suivant
        }
    }
};
```

Exercice 1

1. Ecrire et exécuter ces deux programmes. Que remarquez vous ?
2. Quel est l'intérêt d'utiliser l'interface `Runnable` plutôt que d'hériter directement de la classe `Thread` ?
3. Rappelez la différence entre un thread et un processus.

En outre, l'utilisation des threads impose des mécanismes de synchronisation lors de l'accès à des objets. Un accès *critique* (i.e qui ne peut s'effectuer que par un thread à la fois) est caractérisé par le mot-clef `synchronized` sur un objet. D'autres mécanismes de synchronisation temporelle existent :

– `join` : pour attendre la fin d'un thread ;
– `wait` : le thread qui appelle cette méthode est bloqué jusqu'à ce qu'un autre thread appelle `notify` ou `notifyAll`. Notez que `wait` libère le verrou im-

posé par un accès critique, ce qui permet à d'autres threads d'exécuter des méthodes synchonisées du même objet.
– `notify` débloque un thread bloqué par `wait` (le premier en queue – FIFO) ;
– `notifyAll` débloque tous les threads bloqués par `wait`.

Exercice 2

1. Créer une classe `Compteur` qui gère un compteur et une valeur maximale de sorte que la méthode `compte` de cette classe permet d'incrémenter le compteur jusqu'à ce qu'il atteigne la valeur maximale. La valeur du compteur sera alors affichée
2. Créer la classe `LanceCompteur` qui lance deux threads qui vont appeler la méthode `compte` sur le même objet `Compteur` (on choisira une limite maximale élevée, typiquement 10000000). On attendra que chacun des deux threads aient terminé son exécution en faisant appel à la méthode `join`.
3. Effectuer plusieurs exécutions. Que remarquez vous ? Comment l'expliquez vous ? Comment résoudre ce problème ?

Exercice 3 *Producteur/Consommateur*
Pour mettre en oeuvre un exemple de synchronisation un peu évolué, nous allons considérer un cas d'école : les producteurs/consommateurs (chacun associé à un thread) s'exerçant sur une ressource partagée, une pile FIFO de taille fixe dans cet exercice.

1. On donne le début de la définition de la pile :

```
public class Pile {
    private int[] _stack; // la pile en elle-même
    private int   _size ; // taille maximale de la pile
    private int   _index; // position courante de la dernière position libre

    public Pile(int size) {
        _stack = new int[size];
        _size  = size;
        _index = 0;
    }
    public Pile() { this(5); }

    public boolean isEmpty() { return _index == 0;   } // test de pile vide
    public boolean isFull()  { return _index == _size; } // test de pile pleine
    public int     size()    { return _index;          }
    public int     capacity(){ return _size;           }
```

   Complétez la classe `Pile` en implémentant les deux méthodes synchronisées, `pop` et `push`. Evidemment, on devra s'assurer qu'aucun empilement n'a lieu sur une pile pleine et que, réciproquement, aucun dépilement ne se produit sur une pile vide. Si c'est le cas, on endormira le thread (avec `wait`). L'action réciproque devra donc réveiller l'ensemble des threads en attente : on devra donc utiliser la fonction `notifyAll`.

Maintenant que la ressource partagée est codée, il ne nous reste plus qu'à implémenter les producteurs (classe `Producteur`) et les consommateurs (classe `Consommateur`). Chaque composant devra exercer son activité (empilé ou dépilé) à l'issue d'un temps aléatoire. Dans tous les cas, ces composant partagent certaines caractéristiques : ils travaillent tous sur la même pile `_stack` (la capacité de cette pile est laissée à votre discrétion), utilisent un générateur aléatoire et seront cadencés via un `Thread.sleep` sur une durée maximale de `_delay` ms. Il est donc légitime de définir une classe `ProdConsSharedProperties` contenant ces éléments communs dont les classes `Producteur` et `Consommateur` hériteront.

2. Définir la classe `ProdConsSharedProperties`
3. Un producteur empilera (après un temps aléatoire compris entre 1 et `_delay` ms) des valeurs aléatoires comprises entre 0 et `_MAX` (une variable statique de la classe) qu'on affichera. Définir la classe `Producteur`.
4. Un consommateur dépilera la pile `_stack` après un temps aléatoire (toujours compris entre 1 et `_delay` ms). La valeur dépilée sera affichée. Définir la classe `Consommateur`.
5. Illustrer la validité de votre code en définissant la classe `ProdCons` qui créent deux producteurs et deux consommateurs agissant sur la même pile. Evidemment, aucun phénomène d'inter-blocage ne doit être observé.

Exercice 4 *Un joli chronomètre*
On demande de réaliser, sous forme d'une application graphique utilisant la librairie graphique `swing`, un chronomètre selon les modèles de la figure 1.

FIG. 1 – L'application ChronoApp

On utilisera un thread pour lancer l'incrémentation du nombre de dixième de seconde (ce nombre étant utilisé pour l'affichage de la valeur du chronomètre, au format `h:mm:ss:d`). Un bouton "Go" permettra de lancer le chronomètre (une fois lancé, ce même bouton permettra de stopper le chronomètre pour le relancer ultérieurement). Un bouton "Reset" permettra de remettre le chronomètre à 0.