Chapitre 8

Les Threads en JAVA

Création et exécution de thread Contrôle de l'exécution des threads

1. Introduction

Ne pas confondre multithreading et multiprocessing.

Exécution parallèle = Multiprocessing (process parallèles) => Propriété du Scheduler (ordonnanceur) de l'OS sur un ou plusieurs processeurs par changement de contexte du micro-processeur.
De types d'ordonnanceur :

- ordonnanceur en temps partagé (le plus courant) : gestion par priorité
- ordonnanceur en temps réel : Il assure qu'une certaine tâche sera terminée dans un délai donné (souvent utilisé dans les systèmes embarqués)

Deux JVM java (= 2 processus) s'exécutent donc en multi-processing.

Thread = file d'exécution.

Le multithreading est la capacité d'une même JVM d'exécuter plusieurs tâches en parallèle.
Le multithreading n'est pas propre à JAVA. Il existe aussi dans certains OS.

Le basculement d'un thread à un autre ne nécessite pas un changement de contexte (couteux). On parle aussi de "process léger".
Chacun des threads se partagent un même état du processeur. Le désavantage est que le partage de cette même ressource (dans le cas de la JVM, cet état est tout simplement la mémoire de la JVM, c'est-à-dire les objets) nécessite de synchroniser l'exécution de certaines méthodes. De plus, une mauvaise conception de cette synchronisation peut entrainer un interblocage..

2. Le multithreading en JAVA

Créer un thread en JAVA est une propriété de la JVM et plus précisément une propriété de **l'interpréteur** de P-CODE.

Il est donc facile pour le langage Java d'implémenter les threads qui ne sont donc pas vu comme autant de processus qui s'exécutent en parallèle mais comme un mécanisme de l'interpréteur Java qui interprète les instructions de P-CODE de chacun des threads alternativement par "tranches".

3. Le thread en JAVA

java.lang
Class Thread

java.lang.Object
  └ java.lang.Thread

All Implemented Interfaces:
Runnable

En Java, un thread est un **objet** Java qui est une instance d'une classe qui **hérite de la classe Thread** ou qui **implémente l'interface Runnable**.

3.1. Par héritage

Déclaration du thread :

```
class MonThread extends Thread {

          public void run() {
              // traitement "parallèle"
                . . .
          }
     }
```

Création du thread :

```
 MonThread p = new MonThread ();
 p.start();  → execution de la méthode run
➔ continue à exécuter en //
```

3.2. Par l'interface

Déclaration du thread :

```
class MonRunnable extends Toto implements Runnable {

          public void run() {
              // traitement
                . . .
          }
     }
```

Création du thread :

```
MonRunnable p = new MonRunnable ();
Thread q = new Thread(p);
q.start();
```

Cela veut dire que toute classe peut devenir un Thread du moment qu'elle implémente la méthode run de l'interface Runnable.

4. Exemple

4.1. Par héritage

Création de la classe de définition d'un thread

```
public class SurveillerIP extends Thread
{
   private String adresse;

   public SurveillerIP(String adresse)
   {
      this.adresse = adresse;
   }

   public void run()
```

```
    {
       while(true)
       {
           // Surveillance de adresse
           if (InetAddress.getName(adresse).isReachable(100))
              System.out.println(adresse+" is reachabled");
       }
    }
}
```

Dans du code Java qui veut créer le thread et le lancer :

```
SurveillerIP  surv1 = new SurveillerIP("10.10.1.4");
SurveillerIP  surv2 = new SurveillerIP("10.10.1.5");

surv1.start();
surv2.start();
```

4.2. Par l'interface

Création de la classe de définition d'un thread

```
public class SurveillerIP extends Toto implements Runnable
{
   private String adresse;

   public SurveillerIP(String adresse)
   {
      this.adresse = adresse;
   }

   public void run()
   {
      while(true)
      {
          // Surveillance de adresse
          if (InetAddress.getName(adresse).isReachable(100))
             System.out.println(adresse+" is reachabled");
      }
   }
}
```

Dans du code Java qui veut créer le thread et le lancer :

```
SurveillerIP  surv1 = new SurveillerIP("10.10.1.4");
SurveillerIP  surv2 = new SurveillerIP("10.10.1.5");

Thread t1 = new Thread(surv1);
Thread t2 = new Thread(surv2);

t1.start();  // Elle appelle la méthode run de surv1
t2.start();
```

5. Exemple type d'un thread (Exemple 25 de NSY 102)

Cet exemple représente l'architecture type d'un thread.
Il permet de mettre en évidence les points suivants :

- le traitement cyclique dont chaque itération peut être synchronisé entre les différents threads
- l'usage des attributs static qui permettent de mettre en commun des informations entre les différents threads
- la création des méthodes permettant de suspendre et reprendre l'exécution du thread car les méthodes historiques *suspend* et *resume* sont déprecated
- la façon d'arrêter le thread proprement car la méthode historique *stop* est deprecated
- la mise en veille (sleep) du thread à chaque itération afin d'équilibrer naturellement le partage de l'exécution des threads

Voir sur le site http://jacques.laforgue.free.fr dans les exemples de NSY 102
Exemple25_Thread

6. La synchronisation en JAVA

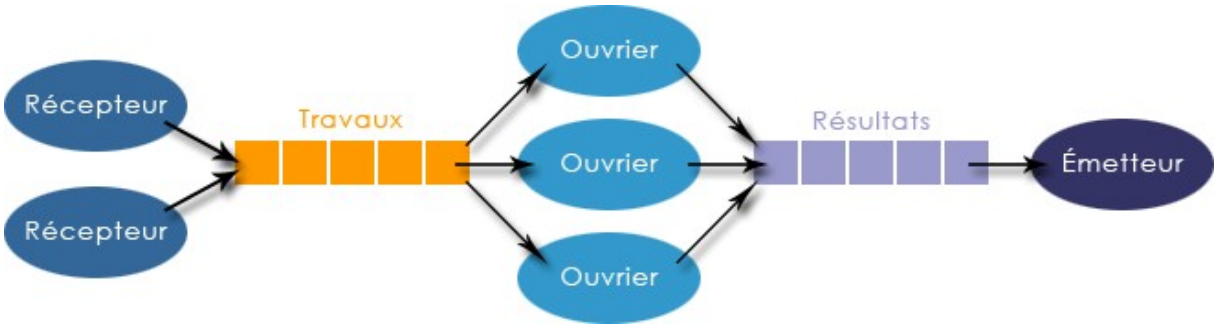
6.1. Introduction

Comme les threads s'exécutent en même temps, des méthodes peuvent appeler en même temps une même portion de code.

Il est souvent indispensable de rendre exclusif l'utilisation de ces portions de code entre les différents threads car leurs usages en parallèle peuvent par exemple mettre en péril l'intégrité d'un objet.

Il est aussi souvent utile d'interrompre l'exécution d'une méthode et la mettre en attente de la libération d'une certaine ressource.

Tout d'abord, voyons un problème de synchronisation classique. Plusieurs entités récepteurs (threads) reçoivent une demande d'un client. Ils peuvent déposer des travaux dans une file. Les ouvriers prennent les travaux déposés, les traitent, et fournissent un résultat dans une autre file. L'émetteur récupère ces résultats et peut les envoyer au client.



Plusieurs problèmes de synchronisation se posent ici :

1/ Plusieurs récepteurs ne peuvent pas déposer simultanément un travail dans la file des travaux, sinon les travaux seraient sur la même case ;

2/ Si la file des travaux est pleine, les récepteurs doivent attendre qu'une case se libère pour déposer un nouveau travail ;

3/ Plusieurs ouvriers ne peuvent pas prendre chacun un travail simultanément, sinon ils auraient le même travail à traiter ;

4/ Si la file des travaux est vide, les ouvriers doivent attendre qu'un travail soit déposé pour pouvoir le traiter ;

5/ Plusieurs ouvriers ne peuvent pas déposer simultanément le résultat d'un travail dans la file des résultats, sinon les résultats seraient sur la même case ;

6/ Si la file des résultats est pleine, les ouvriers doivent attendre qu'une case se libère pour déposer un nouveau résultat ;

7/ Si la file des résultats est vide, l'émetteur doit attendre qu'un résultat soit disponible.

1, 3 et 5 concernent l'exclusion mutuelle
2, 4, 6 et 7 concernent l'attente d'une ressource

Pour mettre en place **l'exclusion mutuelle**, il faut utiliser des **verrous**. Lorsqu'un thread entre dans une section critique, il demande le verrou. S'il l'obtient, il peut alors exécuter le code. S'il ne l'obtient pas, parce qu'un autre thread l'a déjà pris, il est alors bloqué en attendant de l'obtenir. Il est possible d'utiliser un nombre potentiellement infini de verrous, et donc faire des exclusions mutuelles précises : par exemple, **a()** doit être en exclusion mutuelle avec lui-même et avec **b()**, tandis que **c()** doit être en exclusion mutuelle avec lui-même et avec **d()**, mais pas avec **a()** et **b()**...

Pour mettre en place **l'attente d'une ressource**, il faut utiliser des méthodes d'attente (wait) et de libération de ceux qui sont en attente (notify, notifyAll).

Les deux notions marchent ensemble.

6.2. La synchronisation en JAVA

La notion de synchronisation en JAVA repose sur un principe simple : tout objet JAVA peut être utilisé comme un **verrou** logiciel.

Si on regarde la classe Object (dont tous les objets héritent) on a les méthodes :notify, notifyAll, wait.

Ces méthodes permettent de gérer un verrou unique associé à l'objet.

De plus l'usage de **synchronized** permet de rendre exclusif l'exécution d'une portion de code ou de toute une méthode

6.3. Le mot clé Synchronized

Permet de créer une section critique :

```
synchronized(unObjet) {
    //section critique
}
```

unObjet représente un verrou, qui peut être un objet *Java* quelconque. Attention cependant, il vaut mieux utiliser des références déclarées **final**, pour être sûr que la référence vers l'objet n'est pas modifiée. Sauf si on utilise l'objet **this** qui par définition n'est pas déréférencé au moment où on l'utilise.

```
void methode() {
    synchronized(this) {
        //section critique
    }
}
```

est équivalent à :

```
synchronized void methode() {
    //section critique
```

```
}
```

6.4. Exemple (Exemple 26 de NSY 102)

Voir sur le site http://jacques.laforgue.free.fr dans les exemples de NSY 102
Exemple26_Synchronisation

Commentaires :

La méthode **getPremierElementBloquant** retourne le premier élément de la liste ; si la liste est vide, et elle attend qu'il y ait un élément ajouté.

Supposons qu'un thread T1 exécute **ajoute** et qu'un thread T2 exécute **getPremierElementBloquant**.

Supposons que T2 ait d'abord la main, il prend le verrou (la méthode est définie synchronized), il trouve index == 0 vrai, donc il exécute wait(). Ce wait() est bloquant tant qu'un notify() sur le même objet ne le libère pas.

Maintenant, *T1* prend la main. il exécute **ajoute**, et donc demande le verrou. Mais vous allez me dire, il ne l'obtiendra pas, car c'est *T2* qui a le verrou ! Et bien si, il l'obtiendra, car *T2* a lâché le verrou temporairement dès lors qu'il a appelé la méthode **wait()**. Le verrou est quand même toujours "actif" car dès que T2 sera libéré de l'instruction wait, le code reste synchronisé.

T1 peut donc exécuter le code de la méthode **ajoute**. Lorsqu'il exécute **notify()**, il débloque *T2* (de manière asynchrone) de l'instruction wait. Mais, maintenant, bien sûr, *T2* n'a pas le verrou, puisque c'est *T1* qui l'a (pour exécuter le **System.out.println(...)**). *T2* est donc toujours bloqué en attente du verrou, et d'autres threads (imaginons *T3*, *T4*...) peuvent l'obtenir avant lui. Supposons que *T3* supprime tous les éléments de la liste (par une éventuelle méthode **vider()**). Donc, ensuite, *T2* prend la main. T2 est débloqué du verrou. Il a donc besoin de revérifier la condition *index == 0*, sinon il essaierait de récupérer l'élément à l'index 0 du tableau, qui est vide. C'est pour cela qu'il faut utiliser la boucle **while**.

Remarques :

*Si plusieurs threads exécutent **unObjet.wait()**, chaque **unObjet.notify()** débloquera un thread bloqué, dans un ordre indéterminé.*

*L'appel à la méthode **wait()** libère le verrou uniquement parce que l'objet sur lequel a été appliqué **wait()** (ici, **this**) est le même que le verrou.*

*Il ne faut surtout pas confondre cette boucle **while** avec une attente active qui vérifirait en permanence que l'index est différent de 0. Ici, nous regardons si l'index est 0, si c'est le cas, nous mettons le thread en attente passive, qui sera réveillé uniquement lorsqu'un élément aura été ajouté. Une fois qu'il est réveillé, pour la raison que nous venons de voir, nous avons besoin de revérifier la condition.*

Les instructions wait et notify doivent se faire dans des méthodes synchronized.
Ces méthodes sont des méthodes objets d'une instance d'une classe.

Autre exemple très simple :
Deux threads s'exécutent en parallèle sachant que le deuxième thread attend d'être notifier par le premier thread pour continuer son exécution.

Voir le code de Simple.java de l'exemple26