IN328 : *Threads* - Corrigé SI

`http://personnel.supaero.fr/garion-christophe/IN328`

Ce TP est un exemple simple de mise en œuvre des threads et de la synchronisation sous Java.

---

1 Contenu

Ce corrigé succint contient des explications quant aux différentes méthodes de synchronisation à employer dans le TP. Tous les fichiers sources sont disponibles sur le site.

2 Problématique

Cet exercice est tiré de [2]. Ce livre, disponible à la bibliothèque, est d'ailleurs un très bon ouvrage sur la gestion des threads en Java et des problèmes de synchronisation classiques et avancés.

Le problème qui se pose est le suivant : nous allons modéliser les interactions qui s'effectuent dans une cuisine entre deux cuisiniers. Lorsqu'un cuisinier veut faire des cookies, il prend un verre gradué pour mesurer les ingrédients et verse les ingrédients dans un bol. Quand il veut faire une omelette, il prend un bol, bat des œufs, puis mesure la quantité d'œufs dans le verre gradué pour réaliser une omelette pour une personne.

Évidemment, nous supposons que nous sommes dans une cuisine dans laquelle il n'y a qu'un seul bol et un seul verre gradué et qu'il y a deux cuisiniers, un qui veut faire une omelette et un autre qui veut faire des cookies.

3 Modélisation des cuisiniers, du verre gradué et du bol

Les deux cuisiniers seront modélisés par deux threads. Vous pouvez choisir soit d'hériter de la classe `Thread`, soit de réaliser `Runnable`.

Il serait intéressant de pouvoir donner un nom aux threads. Pour cela, on peut utiliser la méthode `setName(String name_)` de la classe `Thread` ou une variante du constructeur de `Thread` (pour récupérer le nom du thread, utiliser la méthode `getName()`. Attention, si vous utilisez `Runnable`, il faudra effectuer ces opérations sur les threads que vous créerez à partir de vos objets de type `Runnable`.

Le bol et le verre gradué seront simplement représentés par deux classes possédant une méthode `utiliser()` (ou `mesurer()`) qui :

- affichera le nom du thread qui prend l'objet ;
- endormira le thread pour un certain temps ;
- affichera le nom du thread qui n'utilise plus l'objet.

4 La classe `Cuisine`

La classe `Cuisine` est le point central du problème. Cette classe devra posséder :

- un attribut de type `VerreGradue` qui représente le verre unique de la cuisine ;
- un attribut de type `Bol` qui représente le bol unique de la cuisine ;
- une méthode `faireCookie()` ;
- une méthode `faireOmelette()`.

Ces méthodes seront utilisées par les cuisiniers qui travaillent dans la cuisine.

5 Implantation et questions

1. écrire les classes `Bol` et `VerreGradue`. Pour les méthodes `utiliser` de ces classes, on affichera des messages explicites quant à l'utilisation du bol ou du verre gradué (de la grande prose du type « le thread X est en train d'utiliser le bol », « le thread X a fini d'utiliser le verre gradué » etc.). On choisira les temps de mise en sommeil représentant l'utilisation de l'objet via un attribut initialisé à la construction des objets.

   Pas de problème particulier pour écrire ces deux classes. Elles devaient juste avoir une méthode `utiliser` qui affiche le nom du thread en train de l'appeler, « endort » le thread, puis réaffiche le nom du thread pour signifier qu'il n'utilise plus l'objet.

2. écrire la classe `Cuisine`. Dans un premier temps, on ne s'occupera pas de problème de synchronisation ;

   La classe qui correspond à cette question est la classe `Cuisine` (disponible sur le site). J'ai également écrit une interface `CuisineInterface` pour pouvoir écrire plus facilement plusieurs versions de la classe.

3. écrire les deux classes représentant les cuisiniers ;

   Là encore, pas de problème particulier. Les classes devaient soit étendre `Thread`, soit réaliser `Runnable`. J'ai pris le parti d'étendre `Thread` (contrairement aux conseils donnés en cours...) pour faciliter la gestion des caractéristiques du thread (nom etc.). Les cuisiniers utilisent l'interface `CuisineInterface` pour pouvoir facilement gérer plusieurs types de cuisine.

4. écrire une classe de test qui crée les threads et la cuisine et met les deux cuisiniers au travail. On choisira les temps de mise en sommeil suivant pour les threads :
   – 1s pour l'utilisation du bol ;
   – 5s pour l'utilisation du verre gradué.
   Que se passe-t-il ? Si l'on pouvait modifier les classes `VerreGradue` et `Bol`, quelle solution simple proposeriez-vous ?

   Le programme de test correspondant est `TestCuisine`. Dans ce cas, on s'aperçoit que le verre gradué est utilisé par Maïté alors que Jean-Paul n'a pas fini de l'utiliser :

   ```
   Le bol est utilise par Maite
   Le verre gradue est utilise par Jean-Paul
   Le bol n'est plus utilise par Maite
   Le verre gradue est utilise par Maite
   Le verre gradue n'est plus utilise par Jean-Paul
   Le bol est utilise par Jean-Paul
   Le verre gradue n'est plus utilise par Maite
   Le bol n'est plus utilise par Jean-Paul
   ```

   Il faut donc trouver une solution pour synchroniser les deux threads. La solution la plus simple serait de déclarer **`synchronized`** les méthodes `utiliser` de `Bol` et de `VerreGradue`. Dans ce cas, les cuisiniers sont obligés d'obtenir un verrou sur le bol ou le verre pour pouvoir les utiliser.

5. nous supposerons dans la suite du TP que nous ne pouvons pas modifier les classes `Bol` et `VerreGradue` (en particulier on peut pas rendre leurs méthodes **`synchronized`**). Une première solution consiste à déclarer un cuisinier « prioritaire » par rapport à l'autre. Le second cuisinier devra donc attendre que le premier ait fini sa recette pour commencer la sienne. Implanter cette solution. Quel impact a-t-elle sur les classes représentant les cuisiniers ? Qu'en pensez-vous ?

La solution pour régler le problème est de rendre Jean-Paul prioritaire par rapport à Maïté. Il suffit pour cela que Maïté fasse un `join` sur Jean-Paul avant de commencer son travail, car l'utilisation du verre gradué prend plus de temps que celle du bol. Le problème est alors résolu :

```
Le bol est utilise par Maite
Le verre gradue est utilise par Jean-Paul
Le bol n'est plus utilise par Maite
Le verre gradue est utilise par Maite
Le verre gradue n'est plus utilise par Jean-Paul
Le bol est utilise par Jean-Paul
Le verre gradue n'est plus utilise par Maite
Le bol n'est plus utilise par Jean-Paul
```

Cette solution est bien évidemment très peu réutilisable : elle dépend du cas particulier présenté ici, en particulier du temps de `sleep` du bol et du verre gradué. Elle nous oblige à écrire des classes `Cuisinier` particulières qui ne conviendront pas pour d'autres utilisations. De plus, le second cuisiner est obligé d'attendre que le premier cuisinier ait complétement fini avant d'utiliser les ustensiles (alors qu'il pouvait par exemple prendre le bol et l'utiliser).

6. on va donc utiliser la possibilité d'obtenir un verrou sur des objets pour résoudre le problème.
   – dans un premier temps, rendre les méthodes de `Cuisine` **`synchronized`**. Que pensez-vous de cette solution ?

     La nouvelle classe « cuisine » est `CuisineSynchro`. Le fait de rendre les méthodes **`synchronized`** ne va pas résoudre le problème puisque l'on pose un verrou sur l'objet de type `Cuisine`. Le premier thread qui obtiendra le verrou utilisera le bol et le verre sans que l'autre thread ne puisse les utiliser. C'est le même cas que le `join` utilisé précédemment :

     ```
     Le bol est utilise par Maite
     Le bol n'est plus utilise par Maite
     Le verre gradue est utilise par Maite
     Le verre gradue n'est plus utilise par Maite
     Le verre gradue est utilise par Jean-Paul
     Le verre gradue n'est plus utilise par Jean-Paul
     Le bol est utilise par Jean-Paul
     Le bol n'est plus utilise par Jean-Paul
     ```

   – proposer une autre solution qui évite d'utiliser la cuisine comme verrou. Qu'en pensez-vous ?

     La solution la plus naturelle est d'utiliser non pas un verrou sur l'objet cuisine en entier, mais d'utiliser des verrous sur les objets de type `Bol` et `VerreGradue`. On pourrait bien sûr écrire la méthode `faireOmelette` de la façon suivante :

     ```
     public void faireOmelette () {
         synchronized(bol) {
             bol.utiliser();
         }
         synchronized(verre) {
             verre.utiliser();
         }
     }
     ```

     Dans ce cas, le problème est résolu. Mais à mon avis, on ne représente pas bien le problème... En effet, pendant qu'il utilise le verre gradué, le cuisinier a besoin d'utiliser le bol. Les méthodes vont donc s'écrire (cf. classe `CuisineSynchroUstensiles`) :

```
public void faireOmelette () {
    synchronized(bol) {
        bol.utiliser();
        synchronized(verre) {
            verre.utiliser();
        }
    }
}
```

Mais dans ce cas, on obtient bien évidemment un interblocage :

```
Maite attend le verrou sur le bol
Maite a le verrou sur le bol
Le bol est utilise par Maite
Jean-Paul attend le verrou sur le verre
Jean-Paul a le verrou sur le verre
Le verre gradue est utilise par Jean-Paul
Le bol n'est plus utilise par Maite
Maite attend le verrou sur le verre
Le verre gradue n'est plus utilise par Jean-Paul
Jean-Paul attend le verrou sur le bol
```

– quelle solution simple pouvez-vous proposer au problème soulevé par la solution précédente ?

Il faut résoudre le problème d'interblocage soulevé par la question précédente. Pour cela, deux solutions s'offrent à nous :

(a) la première solution, la plus couramment utilisée, est de créer une hiérarchie « artificielle » de verrous : on impose que le verrou sur le bol par exemple soit « prioritaire » sur l'utilisation du verrou sur le verre gradué. L'utilisation correcte de cette hiérarchie doit être vérifiée par le programmeur. La classe correspondante est la classe `CuisineSynchroHierarchie`. Le résultat est le suivant :

```
Le bol est utilise par Maite
Le bol n'est plus utilise par Maite
Le verre gradue est utilise par Maite
Le verre gradue n'est plus utilise par Maite
Le verre gradue est utilise par Jean-Paul
Le verre gradue n'est plus utilise par Jean-Paul
Le bol est utilise par Jean-Paul
Le bol n'est plus utilise par Jean-Paul
```

Évidemment, on obtient le même résultat qu'avec le `join` ou les méthodes déclarées globalement **`synchronized`**, mais cette fois-ci on ne bloque pas un verrou entier sur l'objet de type `Cuisine` par exemple.

(b) le problème « classique » de la synchronisation en Java est que le fait de demander un verrou sur un objet est une opération bloquante. On peut utiliser d'autres solutions, comme par exemple créer sa propre classe de verrous pour lesquels le blocage est évité. Nous allons choisir d'utiliser la classe `BusyFlag` présentée dans [2] (disponible sur le site). Cette classe est d'ailleurs un bon sujet d'étude pour comprendre les problèmes de synchronisation. L'utilisation de la classe `BusyFlag` nous permet de tester si le « verrou » est libre (on peut d'ailleurs remarquer que le simple remplacement des **`synchronized`** par des appels à la méthode `getBusyFlag` ne permet pas de résoudre le problème d'interblocage). S'il ne l'est pas,

on peut faire autre chose, verser la farine dans un autre récipient par exemple pour libérer le verre gradué. Ceci n'était pas possible avec les **synchronized**, car on ne libère pas un verrou quand on veut. La classe cuisine correspondante est `CuisineSynchroBusyFlag` et le résultat est :

```
Le bol est utilise par Maite
Le verre gradue est utilise par Jean-Paul
Le bol n'est plus utilise par Maite
Le verre gradue n'est plus utilise par Jean-Paul
Jean-Paul libere le verre gradue en
versant la farine dans un autre recipient
Le verre gradue est utilise par Maite
Le verre gradue n'est plus utilise par Maite
Le bol est utilise par Jean-Paul
Le bol n'est plus utilise par Jean-Paul
```

L'utilisation de cette classe ne nous affranchit pas de tous les problèmes d'interblocages. Nous renvoyons le lecteur à [2] pour plus de compléments.

7. créer deux nouveaux cuisiniers qui doivent faire un gâteau au chocolat. Le premier devra faire fondre le chocolat, le second battre les œufs en neige puis attendre que le chocolat soit prêt avant de mélanger le tout.

   Pas de difficulté particulière ici. Il suffisait de créer deux nouvelles méthodes dans la classe `Cuisine`, une pour le thread qui faisait chauffer le chocolat et deux autres pour le second thread (battre les œufs et mélanger). Il fallait utiliser un `wait` et notifier le thread attendant lorsque le chocolat était prêt. Comme objet sur lequel appeler les méthodes srcnotify et `wait`, j'ai utilisé un `Object syncChocolat` (d'où l'appel non pas à `wait()`, mais à `syncChocolat.wait()` par exemple). La condition pour faire fondre le chocolat est représentée par un booléen `flagChocolat`. Les classes correspondantes sont `CuisineChoco` et `CuisinierChoco`. Le résultat est le suivant :

   ```
   Jean-Paul bat les oeufs
   Jean-Paul attend le chocolat...
   Maite chauffe le chocolat
   Jean-Paul melange le tout...
   ```

   J'utilise également la classe `Console` pour attendre une entrée clavier (pour signifier que le chocolat est fondu). Elle est disponible sur `http://personnel.supaero.fr/garion-christophe/IN201` sous l'onglet « Ressources ».

   Je vous propose également une classe de test JUnit pour tester `CuisineChoco`. JUnit n'est pas prévu pour tester des applications *multithreads*[1], donc j'utilise une extension très propre, MultithreadedTC [1], pour pouvoir tester la classe. Voici le code source de [2] :

   ```
   package fr.supaero.threads;

   import edu.umd.cs.mtc.*;
   import org.junit.*;
   import static org.junit.Assert.*;

   /**
    * <code>MTCChocoTest</code> permet de tester l'interaction entre les
   ```

[1] Essentiellement parce que le *framework* n'attendra pas que les *threads* créés dans le cas de test finissent.
[2] MTCChocoTest

```
 * deux cuisiniers faisant une mousse au chocolat.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class MTCChocoTest extends MultithreadedTestCase {

    private CuisineChoco cuisine;

    public MTCChocoTest() { }

    /**
     * <code>initialize</code> permet d'initialiser le cas de test.
     *
     */
    @Override public void initialize() {
        cuisine = new CuisineChocoWithoutConsole(1000, 5000);
    }

    /**
     * <code>thread1</code> represente le premier cuisinier. Celui-ci
     * attend un premier tick d'horloge avant de faire fondre le chocolat.
     *
     * @exception InterruptedException if an error occurs
     */
    public void thread1() throws InterruptedException {
        waitForTick(1);
        cuisine.chaufferChocolat();
    }

    /**
     * <code>thread2</code> represente le second cuisinier. Celui-ci
     * bat les oeufs, puis appelle la methode melanger. Une fois qu'il
     * l'a appelee, le premier tick d'horloge est lance.
     *
     * @exception InterruptedException if an error occurs
     */
    public void thread2() throws InterruptedException {
        cuisine.battreOeufs();
        cuisine.melanger();
        assertTick(1);
    }

    /**
     * <code>finish</code> permet de verifier un certain nombre d'assertions
     * a la fin de l'execution des threads.
     *
     */
    @Override public void finish() {
        assertTrue(cuisine.getChocolatFondu());
```

```
    }

    /**
     * <code>testMTCChoco</code> est la "vraie" methode de test qui
     * va lancer les threads. On ne les lance qu'une seule fois.
     *
     * @exception Throwable if an error occurs
     */
    @Test public void testMTCChoco() throws Throwable {
        TestFramework.runOnce( new MTCChocoTest() );
    }
}
```

MultithreadedTC nous permet d'utiliser une horloge « virtuelle » pour pouvoir simuler l'ordonnancement de nos *threads* en utilisant les méthodes `assertTick` et `waitForTick`. J'impose dans ce test la chose suivante : la méthode `chaufferChocolat` ne démarrera que lorsque `melanger` aura commencé. Évidemment, le test ici ne devrait pas tout à fait se dérouler comme cela (normalement on ne devrait pas faire appel à l'horloge virtuelle), mais c'est pour vous montrer une utilisation de cette extension.

La syntaxe un peu particulière de la classe provient du fait que MultithreadedTC utilise JUnit 3.8 et non pas JUnit 4, mais vous pouvez utiliser l'archive de JUnit 4 pour la faire fonctionner.

Je vous renvoie à [1] pour plus d'explications.

Références

[1] multithreadedtc : a framework for testing concurrent Java applications. `http://code.google.com/p/multithreadedtc/`.

[2] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 2nd edition, 2000. In French, traduction de V. Lamareille.