TP n°2 : synchronisation de threads en Java.

1 Moniteurs Java

La synchronisation en Java est faite en utilisant les moniteurs. Chaque objet Java a un moniteur (i.e. un `mutex` et une condition) qui lui est associé.

Pour entrer dans un moniteur (i.e. pour prendre le `mutex`), il suffit d'exécuter une méthode qui possède le mot-clé `synchronized`. On sort du moniteur (on relâche le `mutex`) lorsqu'on sort de la méthode. Par conséquent, il n'est pas possible d'exécuter simultanément des méthodes marquées avec le mot-clé `synchronized` sur un même objet.

Le moniteur Java est réentrant. Cela signifie qu'un thread peut réacquérir un `mutex` qu'il possède déjà. En pratique, cela signifie qu'une méthode avec le mot-clé `synchronized` peut appeler une autre méthode du même objet qui possède aussi le mot-clé `synchronized` sans créer un interblocage.

Exercice 1. Créer un package Java nommé `CompteBancaire`. Y placer les classes `Compte` et `Operation` que vous trouverez dans l'archive `compteBancaire.zip`. `Compte` correspond à un compte bancaire et `Operation` correspond à un thread qui effectue des opérations sur un compte bancaire.

1. Examiner le code de ces deux classes et exécuter la classe `Operation`. Constater le problème suivant : `Operation` effectue des opérations qui devraient laisser le solde du compte inchangé, et pourtant, après un moment, le solde ne reste pas à 0 ! Expliquer.
2. Modifier le code pour empêcher ce problème.
3. Dans le code de `Operation`, remplacer la méthode `operationNulle` par deux opérations `ajouter` et `retirer` qui devraient elles aussi laisser le solde du compte à 0. Lancer l'exécution et constater le problème. Modifier le code pour que ça marche.

2 Conditions : `wait` et `notify/notifyAll`

En plus du `mutex` associé à chaque objet en Java, il y a une condition. Les primitives s'appellent `wait(), notify()` et `notifyAll()`. On peut noter qu'elles n'ont pas de paramètre puiqu'elles manipulent le `mutex` et la condition associés à l'objet courant.

2.1 Remarques :

a) lors du réveil, l'ordre FIFO n'est pas garanti,
b) en Java, les moniteurs définissent une file d'attente (une condition) unique,
c) les threads doivent coopérer i.e. si des threads appellent `wait()`, d'autres threads doivent appeler `notify()/notifyAll()`.

2.2 La méthode `wait`

Son fonctionnement est le suivant :

1. nécessite que le thread possède le moniteur,
2. *de manière atomique*, bloque le thread en relâchant le moniteur (comme cela, d'autres threads peuvent l'acquérir), une fois réveillé, réacquiert le moniteur,

L'utilisation correcte de `wait` est généralement la suivante :

```
while (! test) {
      wait();
}
```

L'utilisation avec un `if` est généralement incorrecte puisque, entre le moment où un thread est réveillé et le moment où il réacquiert le moniteur, il est possible que le test soit changé (par ex. parce qu'un autre thread a pris le moniteur entretemps).

3 Utilisation des threads en Java

3.1 Étendre la classe prédéfinie `Thread`

// classes et interfaces prédéfinies Java, JDK

```
interface Runnable {
       void run();
}
public class Thread extends Object implements Runnable {
      void run() { ...  }
      void start() { ... }
      ...
}
public class Compteur extends Thread {
      public class run() { ...  }
      }
      public static main() {
             Compteur c = new Compteur() ;
             c.start();
      }
}
```

L'héritage à partir de `Thread` est contraignant car il empêche tout autre héritage (en Java, une classe ne peut hériter que d'une seule autre classe).

3.2 Utiliser l'interface Runnable

Ceci permet l'héritage d'autres classes et l'implantation d'autres interfaces.

```
interface Runnable {
        void run() ;
}
public class Thread extends Object implements Runnable {
        void run() { ...  }
        void start() { ... }
        ...
}

public class Compteur implements Runnable {
        public void run() { ... }
    }
public static main() {
       Compteur c = new Compteur();
       new Thread(c).start() ;
    }
```

Vous trouverez dans la classe `Répétiteur.java` un exemple d'utilisation de la classe `Runnable`. Exécutez-le au moins 2 fois et observez le résultat.

Exercice 2. Des threads un peu dépendants
Dans la classe `CompteurVarPartagee`, on veut que chaque compteur affiche son ordre d'arrivée. Le message de fin est du type : "`Toto a fini de compter jusqu'à 10 en position 3`". La variable `position` est donc une variable partagée (déclarée `static` en Java).

Exercice 3. Problème des lecteurs-rédacteurs
Dans l'archive `lecteursRedacteurs`, vous trouverez 4 classes. Créer un package `lecteursRedacteurs` et y placer ces classes.
On se donne comme objectif de concevoir une solution Java utilisant la solution native des moniteurs, pour résoudre le problème des lecteurs/rédacteurs.

1. Les classes `Reader`, et `Writer` décrivent les lecteurs et les rédacteurs, qui voudront accéder à un serveur de base de données (instance d'une classe `Database` que vous allez devoir compléter). La classe `Database` devra évidemment synchroniser correctement les lecteurs et les rédacteurs qui tenteront d'accéder à son contenu. Vous en trouverez juste un squelette.
2. Une fois que cela fonctionne, expliquez clairement la stratégie que vous avez mise en œuvre concernant la priorité des lecteurs vis-à-vis des rédacteurs.
3. Essayez ensuite d'implanter une stratégie différente, par exemple, donner priorité aux rédacteurs. Plus précisément, un lecteur ne devra pas se joindre à la session de lecture, même si un lecteur est en train de lire, dès lors qu'un rédacteur attend pour écrire.

Exercice 4. Simutaion rudimentaire de parking
On veut simuler un parking de N places. Chaque voiture a un identifiant (ex. 1,2,3,...) et sera simulée par un thread. Une voiture se comporte comme suit : elle demande à rentrer dans le parking ; quand sa requête est acceptée, elle rentre et attend un temps aléatoire puis elle sort.
Le contrôleur assure que moins de N voitures sont dans le parking. Il fournit deux méthodes : `enter()` qui retourne `true` si la demande d'entrée est acceptée (et `false` sinon) ainsi que la méthode `leave()` qui permet à une voiture d'annoncer qu'elle sort.

Écrire le petit simulateur, une voiture demandera à entrer tant que sa demande sera refusée. Ajouter des messages indiquant ce que font les voitures. Ajouter au contrôleur une liste lui permettant d'afficher régulièrement la liste des voitures garées. Quel est le principal problème de cette implantation ?