TP 2
Threads et Programmation distribuée avec sockets

3 heures

Pré-requis : TP1 terminé

Il arrive souvent que l'on ait plusieurs choses à faire en même temps dans un même programme. Par exemple, dans un client de messagerie, il est courant d'avoir une tâche qui s'occupe de vérifier périodiquement si de nouveaux emails doivent être récupérés pendant qu'une autre tâche s'occupe d'interragir avec l'utilisateur afin de lui permettre de consulter les messages déjà réceptionnés. Dans ce cas, on utilise le plus souvent la programmation multithreads. Conceptuellement, un thread est un morceau de programme qui s'exécute en parallèle d'un autre. Un programme ayant plusieurs threads pourra ainsi donner l'impression de faire plusieurs choses à la fois.

1 Premier threads

Nous allons commencer par construire une petite application multi-thread mettant en évidence un comportement classique en programmation multi-threads.

1. Écrivez une classe *Talkative* qui contient un constructeur prenant en paramètre un entier qui est un attribut de la classe ;
2. Modifiez votre classe afin qu'elle implémente *Runnable* ;
3. Redéfinissez la méthode *run* afin qu'elle affiche 100 fois la valeur de l'attribut contenu par la classe ;
4. Dans une méthode statique *main* créez 10 instances de la classe *Thread*. Chacune de ces instances prendra en paramètre une nouvelle instance d'un objet de type *Talkative*. Chacun des objets de type *Talkative* doit être construit en prenant en paramètre un entier unique ;
5. Appelez la méthode *start* sur chacun des objets de type *Thread* qui ont été créé.
6. Que constatez vous ?

2 Situation de compétition

Le comportement que vous avez pu constater dans l'exercice précédent mène à des situations de compétition (*race conditions* en anglais). Cela se produit lorsque deux threads ou plus tentent de modifier en même temps une variable partagée. Le but de cet exercice est de reproduire un tel cas et de comprendre comment il peut être corrigé.

1. Créez une nouvelle classe nommée *SimpleRaceCondition* ayant une variable d'instance privée de type *int* nommée *c* ;

2. Ajoutez deux méthodes d'instance, publiques, sans valeur de retour ni paramètre, nommées respectivement *increment* et *decrement*. La première méthode est sensée incrémenter la variable c de 1 tandis que la second méthode doit la décrémenter de 1 ;

3. Ajoutez une nouvelle méthode ayant la signature `public void doSomeWork()`. Cette méthode doit dans son corps créer et démarrer deux threads, chacun prenant en paramètre une instance de *Runnable* dont la méthode *run* aura été redéfinie. Pour le premier thread, la méthode *run* doit être redéfinie afin de faire appel à la méthode *increment* 10^4 fois tandis que le second thread doit redéfinir la méthode *run* afin de faire appel à la méthode *decrement* 10^4 fois ;

4. Toujours dans la méthode *doSomeWork*, après avoir démarré les threads, forcez la méthode à attendre la fin de l'exécution des threads en appelant la méthode *join* sur chacun des deux threads ; à la suite de cela ajoutez une trace permettant d'afficher la valeur de la variable c sur la sortie standard ;

5. Finalement, ajoutez une méthode *main* statique à la classe *SimpleRaceCondition*. Cette dernière doit créer une instance de la classe et appeler la méthode *doSomeWork* ;

6. Exécutez votre application. Quelle est la valeur de la variable c à la fin de l'exécution ? la valeur doit être différente de 0, pourquoi ?

7. Proposez une solution permettant de corriger la situation de compétition (e.g. en utilisant le mot réservé `synchronized`). Appliquez la et vérifiez qu'elle est valide.

3 Producteurs/consommateurs

L'objectif de cet exercice est d'introduire le mécanisme d'attente passif de Java via les méthodes *wait* et *notify* dont tout objet Java dispose. Pour cela nous nous proposons de créer un scénario de producteurs/consommateurs partageant un buffer borné à une seule place. Il s'agit pour les producteurs et les consommateurs (qui seront exécutés dans des threads indépendants) de se synchroniser sur le buffer. Un producteur doit attendre que le buffer soit vide pour mettre la valeur produite dans le buffer. Un consommateur attend que le buffer soit plein pour consommer la valeur.

1. Créez une classe *Buffer* en accord avec l'énoncé. C'est-à-dire qu'elle doit contenir une variable d'instance de type *Integer* et deux méthodes :

   (a) La première nommée *get* est sans paramètre mais retourne la valeur contenue dans le buffer puis réveille les threads en attente d'écriture dans le buffer (via un appel à la méthode *notifyAll* de l'instance courante). Dans le cas ou la valeur du buffer est nulle, l'appel à cette méthode doit mettre le thread courant en attente (via un appel à la méthode *wait* sur l'instance courante) jusqu'à ce qu'un producteur ajoute une une nouvelle valeur au buffer.

   (b) La seconde méthode nommée *put* qui prend en paramètre un *Integer* doit affecter la valeur au buffer si il est vide et réveiller les threads en attente. Cependant si le buffer est plein, l'appel à cette méthode doit bloquer le thread courant jusqu'à ce qu'une place dans le buffer se libère.

2. Créez une classe *Producer* avec un constructeur qui prend en paramètre un objet de type *Buffer* et un entier primitif représentant l'identité du producteur. Cette classe doit étendre

la classe *Thread* et redéfinir la méthode *run* afin d'ajouter 100 fois une valeur dans l'instance du buffer contenue par la classe. Chaque appel sur le buffer étant séparé de quelque millisecondes via l'instruction `Thread.sleep((int)(Math.random() * 100));` et d'une trace qui affiche sur la sortie standard l'identité du producteur ainsi que la valeur mise dans le buffer.

3. Créez une copie de la classe *Producer* et nommée la *Consumer*. L'entier pris en paramètre identifie maintenant un consommateur. La méthode *run* doit être modifiée afin de récupérer les valeurs contenues dans le buffer et la trace éditée afin d'afficher la valeur récupérée.

4. Enfin, créez une classe *Main* contenant la méthode suivante :

```
public static void main(String[] args) {
    Buffer buffer = new Buffer();
    Producer p1 = new Producer(buffer, 1);
    Producer p2 = new Producer(buffer, 2);
    Producer p3 = new Producer(buffer, 3);

    Consumer c1 = new Consumer(buffer, 1);
    Consumer c2 = new Consumer(buffer, 2);
    Consumer c3 = new Consumer(buffer, 3);

    p1.start(); p2.start(); p3.start();
    c1.start(); c2.start(); c3.start();
}
```

5. Lancez l'application et vérifiée que l'application fonctionne correctement.

4 Gérer plusieurs clients

Le serveur écrit durant le premier TP est très limité car il ne sait gérer qu'un seul client à la fois. Un cas d'utilisation plus réaliste serait de pouvoir traiter un client (lire/écrire), tout en en acceptant d'autres (accept). Pour réussir cela il faut utiliser des *threads*.

1. Pour gérer plusieurs clients il faut écouter en permanence les connexions et démarrer un nouveau thread dès qu'un client arrive, pour le traiter ;

2. Créez une classe *ClientManager* qui, grâce à une socket passées en paramètre de son constructeur, s'occupe du dialogue avec un client ;

3. Créez une copie de la classe *Server* écrite lors du premier TP et nommée la *MultithreadedServer* ;

4. Modifiez la classe *MultithreadedServer* pour que lorsqu'une connexion arrive, il crée une nouvelle instance de *ClientManager* et appelle la méthode responsable du dialogue ;

5. Testez que tout fonctionne comme auparavant ;

6. Modifiez votre classe *ClientManager* pour qu'elle implémenter *Runnable* et redéfinissez la méthode *run* afin qu'elle appelle la méthode responsable du dialogue ;

7. Modifiez votre serveur pour qu'il accepte plusieurs clients. Il faut pour cela faire plusieurs appels à la méthode *accept* sur l'objet *ServerSocket*. De plus, votre serveur doit démarrer un nouveau thread lorsqu'une nouvelle connexion arrive. Ce thread doit être construit en passant au constructeur une instance de *ClientManager* ;

8. Testez que tout fonctionne correctement en lançant 3 clients différents.