Ateliers java - séance 10

Objectifs :

→ Être capable de créer, de lancer et d'arrêter proprement des threads
→ Être capable d'écrire du code thread safe
→ Maîtriser les concepts avancés de la concurrence

Thèmes abordés :

- Concurrence, Concurrence avancée

Récupérez les sources de la séance 10 et importez le projet dans votre workspace.

1. Concurrence-Chengdu

La `Bourse` au minerai de Chengdu est la plaque tournante mondiale de l'échange du scandium. Elle est entièrement virtualisée et fonctionne de la façon suivante :

Des `Lot`s de minerai sont mis en vente par des vendeurs. Pour chaque `Lot` on retient sa quantité, s'il est en vente ou pas et un prix courant de mise en vente. Il contient également un numéro de lot. A chaque `Lot` est toujours associé un et un seul propriétaire qui est une `Personne` dont on retiendra le nom et le solde de son compte en banque. Pour chaque `Personne` on retient les `Lot`s dont il est propriétaire.

Les ventes se font un peu à la manière des enchères Hollandaises : le `Lot` est d'abord placé à un prix élevé. Ensuite, ce prix est diminué à intervalle de temps régulier jusqu'à ce qu'il trouve un vendeur ou qu'il atteigne un prix plancher.

La classe `Bourse` est un singleton implémenté à l'aide d'un `enum`. `Elle` stocke tous les `Lot`s en vente.

La vente d'un lot

Lorsqu'une personne désire mettre en vente un `Lot`, elle lance un robot `Vendeur` (un thread java en fait). La personne indique à son robot le `Lot` à mettre en vente, le prix initial de vente du `Lot`, le montant de la diminution, le prix plancher et l'intervalle de temps entre deux diminutions de prix (en millisecondes).
Ce `Vendeur` va placer dans le système de bourse le `Lot` et va ensuite se réveiller à intervalle régulier. Si l'objet n'est plus en vente, le thread s'arrête. S'il est toujours là, le prix de l'objet est diminué à condition que celui-ci reste bien supérieur ou égal au prix plancher. Dans le cas contraire, l'objet est retiré de la vente et le thread se termine.

C'est la classe interne `Vendeur` (implémentant `Runnable`) de la classe `Personne` (package `chengdu.domaine`) qui permet de réaliser cela.

Le constructeur de la classe `Vendeur` reçoit en paramètres tous les mêmes paramètres que la méthode `mettreUnLotEnVente`. La vérification des paramètres est déjà faite sauf celle qui consiste à vérifier que le lot mis en vente appartient bien à la personne voulant le mettre en vente que vous devez ajouter.
L'initialisation des attributs et la mise en vente du lot (`setEnVente(true)`) sont déjà implémentées.

La méthode `run` de la classe `Vendeur` fait ce qui est indiqué ci-dessous :

1. d'abord fixer le prix unitaire du lot ;
2. ajouter le lot à la bourse (afficher le texte : `Mise en vente du lot numéro 1 au prix de 10.0` par exemple) ;
3. attendre un certain temps ;
4. diminuer le prix ;
5. refaire les étapes 3 et 4 tant que le thread n'est pas interrompu, que le lot est encore en vente et que le prix reste supérieur ou égal au prix plancher. Quand on ne peut plus diminuer son prix, le lot est retiré de la bourse et on déclare qu'il n'est plus en vente (afficher le texte : `Lot numéro 1 retiré de la vente`).

Complétez les TODO de la méthode run.

Ensuite, complétez la méthode `mettreUnLotEnVente` afin qu'elle crée un `Thread` pour un vendeur et qu'elle le démarre.

L'achat d'un lot

Lorsqu'une personne désire acheter du minerai, elle lance un thread `Acheteur`. Elle doit lui indiquer la quantité de minerai à acheter et le prix maximum par unité de minerai qu'elle est prête à payer. Ce robot va regarder l'ensemble des lots en vente. Lorsqu'il rencontre un `Lot` qu'il peut acheter (il n'est pas le vendeur du lot, le prix par unité est inférieur ou égal au prix maximum, la quantité mise en vente est inférieure à la quantité restante à acheter, le solde de l'acheteur est suffisant pour l'acheter) alors il l'achète. Il est obligatoire d'acheter des lots entiers.
Le thread `Acheteur` est également une classe interne de `Personne`. On vous demande de l'implémenter.

Pour chaque `Lot` acheté, les actions suivantes sont effectuées (pas forcément dans cet ordre-là) :

1. l'objet est retiré de la vente ;
2. le montant de la transaction est débité du compte de l'acheteur et crédité sur le compte du vendeur ;
3. l'objet est retiré de la liste du vendeur et est ajouté à la liste de l'acheteur ;
4. la quantité restante est diminuée de la quantité achetée (le thread est arrêté si la quantité restante passe à 0) ;
5. un message de type "`Achat du lot numéro 1 par Emmeline au prix de 7.0`" est affiché.

Après avoir parcouru tous les lots en vente, le thread `Acheteur` s'endort pour une durée aléatoire comprise entre 0 et 1000 millisecondes (il s'agit d'un règlement de la commission boursière de Chengdu afin d'assurer la « fairness » de la bourse) avant de les parcourir à nouveau.

Il faut également faire en sorte que le `Thread` s'arrête proprement.

Chaque personne peut créer un nombre arbitraire d'`Acheteur`s et de `Vendeur`s.

La synchronisation des classes

Vous devez faire particulièrement attention à éviter les conditions de course (race) et les deadlocks. Pour cela vous devez maintenant rendre votre code thread safe en utilisant la synchronisation vue dans la théorie « concurrence ». Chaque thread ne peut évidemment pas s'accaparer tout le système.

2. Concurrence avancée

2.1. Opération-Compte

Que fait la classe `Operation` du package `operation_compte` ?

Exécutez cette classe et répétez cette exécution jusqu'à ce que la somme finale ne soit pas égale à 0. D'après le code, ceci ne devrait jamais arriver ! Nous avons donc un problème de concurrence. Changez la classe `Compte` pour résoudre ce problème.
Note : le `Thread.sleep` de la classe `Compte` est présent en vue de favoriser le problème de concurrence. Laissez-le en place. Si vous l'enlevez, il sera beaucoup plus difficile de reproduire le problème, ce qui ne signifie pas que le problème a disparu bien sûr !

2.2. Producteur-Consommateur

Le code de cet exercice est un producteur consommateur implémenté à l'aide d'une `BlockingQueue`. Cependant il y a un point de la classe `Message` qui est faible : elle n'est pas immuable. Le message envoyé est mis en place par le constructeur et reste ensuite immuable. Par contre la réponse est implémentée par une `LinkedBlockingQueue` et on pourrait appeler le setter et le getter plusieurs fois sur le même objet pour transmettre des valeurs différentes. Corrigez l'implémentation de Message pour empêcher cela. L'astuce est de voir que la réponse reste immuable si les deux conditions suivantes sont vérifiées :

- On ne peut appeler qu'une seule fois le setter, les autres fois une exception est jetée.
- Chaque appel du getter retourne la même réponse.

Pour remédier à ce problème, il faut faire en sorte que :

- La première fois qu'on appelle le setter (on doit envoyer la réponse sur la queue) est différentes des fois suivantes (on jette une exception de type `UnmodifiableSetException` pour empêcher l'opération).
- La première fois qu'on appelle le getter (on doit récupérer la réponse de la queue) est différente des fois suivantes (on se contente de renvoyer le résultat déjà obtenu).

De quelle manière allez-vous pouvoir faire cette distinction ? Plus exactement, de quelle manière **atomique** ? Un bon point de départ pour répondre à cette question est la Javadoc du package `java.util.concurrent.atomic` :
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

2.3. Écriture dans un fichier

Regardez la classe `Fichier`. Cette classe implémente un tableau de 100 bytes stocké dans un fichier.

1. Créez une classe de test qui instancie un objet `Fichier`. Cette classe doit créer 100 threads. Chaque thread connaît son propre numéro (de 0 à 99) et écrit dans le fichier à la position de ce numéro le numéro lui-même. Chaque thread répète cette opération 25 fois.
2. À la fin le fichier doit donc contenir les bytes 0, 1, 2, …, 99 dans cet ordre. Complétez votre classe de test afin qu'elle lise et affiche le contenu du fichier.
3. Le fichier n'est pas du tout correct ! Pourquoi chaque thread répète-t-il l'opération 25 fois ? Essayez avec 1, 3 et 10 fois.
   Corrigez cela en faisant en sorte que le `seek` ne se fasse plus à l'extérieur des méthodes `get` et `set`.
4. Modifiez votre programme de test et testez à nouveau la classe Fichier.
5. Le fichier n'est toujours pas correct ! Le problème, c'est que, quand les threads appellent des set en parallèle, les seeks se mélangent quand même les pinceaux. Corrigez cela à l'aide d'un `LinkedBlockingQueue` : les méthodes exposées par `Fichier` émettent des messages sur cette liste tandis qu'un seul thread lit ces messages et les exécute. N'oubliez pas que la méthode get doit aussi être corrigée en utilisant cette technique.
6. Testez de nouveau le programme, le résultat est maintenant correct.