Travaux pratiques :

Mon package concurrent

Comme nous l'avons vu (ou allons le voir) en cours, le package java.util.concurrent introduit différents outils efficaces pour synchroniser les threads ou encore gérer leur exécution. De cette façon, vous n'aurez surement jamais à développer vos propres implémentations de ces mécanismes. Connaitre l'existance de ces outils et savoir les utiliser de façon pertinente est déjà une bonne chose en soit. Cependant, il peut être très utile, en pratique, de connaitre la théorie de ces mécanismes. Ce TP vous propose d'implémenter vos propres version de certaines classes proposées par le package java.util.concurrent. Pour cela, commencer par créer un nouveau package myConcurrent*, puis implémentez et ajoutez y chacune des classes ci-dessous (N'hésitez pas à les garnir de méthodes personnelles suivant vos envies).*

Pour se faire la main

Exercice 1
Un **verrou** est un mécanisme permettant de restreindre l'accès à une section critique : il permet de garantir qu'un seul thread à la fois exécute le code critique.

Un verrou de base offre deux méthodes : une première méthode *lock()* permettant de prendre le verrou, une seconde méthode *unlock()* permettant de relacher le verrou.

Petit complément de cours : Pour des raisons inexpliquées, il arrive qu'un Thread sorte d'un état *wait* sans avoir reçu de signal *notify*. Ce problème est connu sous le nom de **spurious wake up**. Ainsi, lorsqu'un Thread est amené à s'endormir suite à une condition, il convient de vérifier à nouveau la condition lorsqu'il se réveille, ce afin de le rendormir si il n'était pas temps pour lui de se remettre au travail.

Un petit mécanisme, baptisé **spinlock**, permet de réaliser rapidement et efficacement ce garde fou. Le principe de ce dernier est simplement de remplacer le code *if(Condition)* { *wait()* } par *while(Condition)* { *wait()* }. Ainsi un Thread se réveillant vérifiera à nouveau la condition avant de poursuivre sa tâche ou se rendormir.

Quelques indications :
- utilisez un *flag* pour connaitre l'état du verrou, un booléen par exemple ;
- pensez à synchroniser les appels aux méthodes ;
- que faire lorsqu'un thread exécute la méthode *lock()*, alors que le verrou est déja réservé ?
- le verrou doit propager les exceptions, non les capturer !
- testez votre classe avec l'exercice du compteur.

Compliquons un peu

Exercice 2
Un **sémaphore** est un mécanisme de synchronisation entre threads. Il permet de définir des *points de rendez-vous*, utiles pour propager des messages ou garantir un accès restreint à une section critique.

Un sémaphore se comporte comme une boite de jetons et offre, de base, deux méthodes : une méthode *acquire()*, permettant d'obtenir un jeton, et une méthode *release()*, permettant de rendre un jeton.

quelques indications :
- le constructeur permettra d'initialiser la taille de notre boite à jeton, cette taille sera reçue en paramètre ;
- que faire lorsque d'un thread qui demande un jeton, alors que a boite est vide ?

Exercice 3
Une **file bloquante** est une file capable de bloquer lorsqu'on tente de retirer un objet alors qu'elle est vide, ou lorsqu'on tente d'enfiler un objet alors qu'elle est pleine. Un thread qui tente de retirer un objet d'une file vide est mis en attente jusqu'à ce qu'un autre thread dépose un objet. De la même façon, un thread tentant de déposer dans une file pleine est mis en attente jusqu'à ce qu'une place se libère.

Une file bloquante offre, de base, une méthode *enqueue()* permettant de déposer un objet, et une méthode *dequeue()* permettant de retirer un objet.

quelques indications :
- une file bloquante doit encapsuler une file classique, utilisez l'implémentation de l'interface *Collection* qui vous plait le plus ;
- la taille de la file sera reçue en paramètre par le constructeur ;
- que faire d'un thread qui tente de récupérer dans une file vide, ou d'enfiler dans une file pleine ?
- que doit faire lorsqu'un thread tente de déposer dans une file qui est vide, ou à l'inverse tente de retirer d'une file pleine ?
- Pensez à tester votre classe avec l'exercice des producteurs/consommateurs.

Pour aller plus loin

Exercice 4
Un **verrou réentrant** permet au possesseur d'un verrou de demander le verrou à nouveau. En effet, observez votre classe *verrou* : supposons qu'un thread obtienne le verrou et execute la section critique. Supposons que dans cette section critique, il execute une méthode demandant à nouveau le verrou. Ce thread restera alors bloqué à attendre qu'il libère le verrou : il y a *deadlock* ! Comment modifier votre classe *verrou* pour obtenir une classe *verrouReentrant* ?

quelques indications :
- pensez à mémoriser l'identité thread qui obtient le verrou. que faut il faire lorsque ce thread se présente à nouveau pour obtenir le verrou ?
- Quand faut il définitivement libérer le verrou ? Pensez à compter le nombre d'accès au verrou pour le thread courant.

Exercice 5
Un **verrou lecture/écriture** se comporte comme un double verrou. Cependant, ces verrous doivent interagir de la façon suivante :

- il n'est possible d'obtenir le verrou en écriture que si aucun thread n'est en train d'écrire ou de lire la donnée critique ;
- il n'est possible d'obtenir le verrou en lecture que si aucun thread n'écrit ou n'attend en attente d'écrire la donnée critique.

Votre verrou en lecture proposera par défaut quatre méthode de base : *lockRead()*, *unlockRead()*, *lockWrite()* et *unlockWrite()* dont le comportement est explicite.

indication : il peut être utile de compter le nombre de threads en attente du verrou en écriture.

Exercice 6
Les **pool de threads** sont très utiles lorsque l'on souhaite limiter le nombre de threads déployés par l'application. La création d'un thread est en effet couteuse en ressources. De plus, dans le cadre d'un serveur multithread, il peut être souhaitable de vouloir limiter le nombre de threads clients.

Au lieu de créer un nouveau thread pour chaque nouvelle tâche, la tâche est mise en attente en attendant qu'un thread disponible puisse l'exécuter. De façon interne, les tâche sont accumulées dans une *file bloquante*. Dès qu'une tâche est insérée dans la file, un thread la récupère et l'exécute. Les autres threads attendent que de nouvelles tâches soient enfilées.

De base, un pool de thread fournit une méthode *execute(Runnable task)*, qui permet d'enfiler l'execution d'une tâche, ainsi qu'une méthode *stop()* permettant de propager une terminison à l'ensemble des threads gérés.

quelques indications :

- il faudra donc gérer une collection de threads en plus d'une file de tâches ;
- le constructeur recevra en paramètre la taille de la file de tâche, ainsi que le nombre de threads souhaités ;
- il vous faudra surement implémenter une surcharge de la classe thread afin de pouvoir répondre à divers besoins spécifiques ;
- utilisez un *flag* pour connaitre l'état de votre pool, actif ou arrêté ;
- demander l'exécution d'une tâche à un pool stoppé (ou en cours d'arrêt) doit provoquer le lancement d'une exception.