TD 2 : concurrence, sections critiques

Exercice 1 :

Le code td2cours.jar sur le serveur commun simule l'inscription à un cours par plusieurs threads en parallèle. Un cours ayant un nombre maximum de participants, un test dans la méthode

public void inscrire(int nbPlaces) throws PasAssezDePlacesException

vérifie qu'il y a assez de places avant de valider l'inscription, sinon lève une exception.

1.1 Thread-safety dans la ressource

La classe Cours n'est pas thread-safe (elle est algorithmiquement valide mais les conflits liés au multi-threading ne sont pas prévus). Nous allons donc résoudre ce problème.

Repérer la ressource partagée et les runnables qui partagent cette ressource
Comprendre comment un problème peut se poser (trop de participants vont être inscrits)
Provoquer à l'exécution ce problème par un Thread.sleep(2) placé judicieusement (le problème peut apparaître sans le ralentisseur mais c'est peu probable)

Il ne reste plus alors qu'à protéger par un verrou les sections critiques dans le code de la ressource et vérifier que le problème est résolu (toujours avec le ralentisseur bien sûr !)

1.2 Thread-safety dans l'accès à la ressource

Le travail qui vient d'être fait n'est pas envisageable si vous ne disposez pas du code source de Cours. Dans ce cas, on gèrera la thread-safety dans les runnable.
Regroupez Cours et PasAssezDePlacesException dans un package cours.
Exportez ce package sous forme d'un fichier .jar contenant uniquement les fichiers compilés (.class)
Supprimez le package de votre projet
Associez le .jar créé à votre projet (Projet>>Propriétés>>JavaBuild path>>Add jar externe

Reprenez l'exercice en travaillant sur le code de la classe du runnable Inscription : le ralentisseur n'étant pas envisageable, que peut-on faire si le problème de thread-safety n'apparait directement pas aux tests ?

Exercice 2 :

L'application td2incr.jar sur le serveur commun produit des résultats faux et irréguliers (la valeur finale devrait être affichée en dernier et être 21).

En faisant varier le Thread.sleep() du main (ou en le retirant) on voit tout de suite le 1er problème.
Le 2ème problème devrait apparaître tout seul au bout de quelques essais.

Nous allons résoudre les 2 problèmes.

2.1. Afficher la valeur finale quand on est sur que les 2 incrémenteurs ont terminé

Ce problème relève de la synchronisation entre threads.
Vous retirerez le Thread.sleep() du main et introduirez à la place une attente explicite de la fin des 2 incrémenteurs dans le main via la méthode *join()* de la classe Thread[1].

2.2. Obtenir 21 de manière fiable

Ce problème est un problème de thread-safety.
En faisant différents essais, vous devez obtenir une valeur finale qui est variable (entre 11 et 21). Il y a visiblement parfois des incrémentations qui ne sont pas comptabilisées. Ce problème est lié à une utilisation d'une ressource partagée sans gestion des conflits (donc non thread-safe).

[1] unThread.join() met en veille le thread courant jusqu'à ce que l'exécution de unThread soit terminée

Pour rendre l'exécution thread-safe, il faut :

- identifier la ressource partagée et les runnable qui partagent cette ressource ;
- repérer précisément la (ou les) section critique ;
- verrouiller la ressource partagée pendant la section critique ;

Pour ce dernier point, il vous faudra trouver une solution particulière, la ressource partagée n'étant pas ici un objet et ne possédant donc pas de moniteur de Hoare.

2.3 Généralisation de la ressource

Ecrire un runnable Decrementeur et faire agir des incrémenteurs et des décrémenteurs sur un cpt commun. Vérifiez les résultats obtenus.