TP n° 2.

Pour ce TP, récupérer l'archive TP2.zip. Vous y trouverez un certain nombre de classes utiles pour les exercices.

Exercice 1. Des threads indépendants
Soit la classe `Compteur` où l'objet `Compteur` a un nom et il compte de 1 à n, ($n > 0$). Il marque une pause aléatoire entre chaque nombre (de 0 à 5000 ms par exemple).

Un `Compteur` affiche chaque nombre (`Toto` affichera par exemple "`Toto : 3`" et il affiche un message du type "*** Toto a fini de compter jusqu'à 10" quand il a fini.

Exécuter la classe `Compteur` et observer ce qui se passe.

- Remarquer que chaque thread boucle, et à chaque tour de boucle, appelle la méthode `Thread.sleep`. Lors de ces appels système, le thread se voit retirer la CPU (puisqu'il demande à dormir !) afin de faire avancer un autre thread prêt. L'alternance entre les 4 threads est-elle *parfaite* ?
- Mettre l'appel à `sleep` en commentaire et tester.
- Diminuer la valeur 1000000 qui apparaît dans le code source, et la mettre par exemple à 1000. Que constatez-vous ? Expliquer.

Exercice 2. Des threads un peu dépendants
Modifier la classe `Compteur` pour 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`".

Exercice 3. Vous trouverez dans l'archive `TP2.zip` la classe `Compte` qui correspond à un compte bancaire et la classe `Operation` qui 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 l'opération `nulle` par 2 opérations `ajouter` et `retirer` qui devraient elles aussi laisser le solde du compte à 0 (elles sont en commentaire dans le code). Lancer l'exécution et constater le problème. Modifier le code pour que ça marche.

Exercice 4. Problème des lecteurs-rédacteurs
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.