Verrou et attente passive

Exercice 1 - Producteur et Consommateur

On souhaite écrire un programme permettant de simuler un comportement de producteurs/consommateurs.
Un producteur est une thread qui à interval régulier va aller stocker dans une file bloquante un message.
Un consommateur est une thread qui a interval régulier va aller chercher un message dans la file bloquante et afficher le message.
On entend par liste bloquante une liste qui si il n'y a plus de message met les consommateur en attente si ceux-ci demande un message et les reveils lorsqu'un message arrive et met les producteurs en attente si la liste est pleine et les reveils lorsque un message est lu.

1 Nous allons utiliser un `java.util.concurrent.ScheduledExecutorService` pour ordonancer dans le temps les producteurs et les consommateurs.
Rappeler la différence entre les méthodes `schedule`, `scheduleAtFixedDelay` et `scheduleAtFixedRate` de `ScheduledExecutorService`.
Puis, créer un `ScheduledExecutorService` à partir de la classe `Executors` et créer une tâche qui affiche bonjour toutes les 300 millisecondes.

2 Créer une méthode créant le `Runnable` d'un producteur avec en paramètre la chaine de caractère qui sera inséré par le producteur dans la file bloquante.
Pour implanter la file bloquante, utiliser la classe `java.util.concurrent.LinkedBlockingQueue` ainsi que ses méthodes `put()` et `take()`.

3 Faire la même chose avec un consommateur en prenant en paramètre un entier unique `id`. Le consommateur après avoir retier le message de la file bloquante affichera celui-ci avec son numéro unique`id`.

4 Expliquer pourquoi si on plus de consommateurs que la taille core pool thread du scheduled executor, il peut y avoir un deadlock.

5 Tester votre programme dans le cas où la liste bloquante est pleine ainsi que dans le cas où la liste bloquante est vide.

Exercice 2 - File bloquante synchronizée

On souhaite maintenant écrire notre propre classe de liste bloquante appelée `SynchronizedBlockingQueue` et basé en interne sur une `LinkedList`.

1 Écrire un constructeur prenant en paramètre la capacité maximal de la file bloquante.

2 Écrire deux méthodes privées `isFull` et `isEmpty` testant si la file est pleine ou vide.

3 Écrire les méthodes `put` et `take` qui insére un message dans la file et bloque si la file est pleine et respectivement enlève un message de la file et bloque si la file est vide.
Vous utiliserez la méthode `Object.wait()` pour effectuer l'attente. Attention au `spurious wakeup` !
Pour réveiller les threads en attente, comparer les méthodes `Object.notify()` et `Object.notifyAll()` et indiquer celle qu'il faut mieux utiliser ici.

4 Comment faire pour vérifier que les méthodes `isFull` et `isEmpty` sont appelées dans un contexte synchronizé ?

Exercice 3 - File bloquante verrouillée

On souhaite maintenant une autre implantation de liste bloquante appelée `LockedBlockingQueue` utilisant les `java.util.concurent.locks.Lock` à la place des blocs synchronizées.

1 Comment peut-on créer une condition à partir d'un `Lock` ?
2 Quel est l'intérêt d'utiliser des conditions plutôt que des `wait/notify` ?
3 Comment faire pour vérifier que les méthodes `isFull` et `isEmpty` sont appelées entre les méthodes `lock` et `unlock` ?
4 Écrire les méthodes `put` et `take` de la classe `LockedBlockingQueue`.