

Accès aux variables partagées, atomicité

Exercice 1 - Modification d'une variable en concurrence

On souhaite créer deux threads qui change le même un champs d'un même objet :

```
public class Test {
    int value;

    public static void main(String[] args) {
        final Test test=new Test();

        for(int i=0;i<2;i++) {
            final int id=i;
            new Thread(new Runnable() {
                public void run() {
                    for(;;) {
                        test.value=id;
                        if (test.value!=id)
                            System.out.println("id "+id+" "+test.value);
                    }
                }
            }).start();
        }
    }
}
```

- 1 Qu'affiche le code suivant avec la ligne de commande : `java Test` ?
Expliquer.
- 2 Qu'affiche le même programme mais avec la commande `java -server Test`
Expliquer.
- 3 Comment doit on faire pour être sûr que chaque thread voit les modifications effectués sur une variable par l'autre thread ?

Exercice 2 - strtok

On souhaite écrire une version thread-safe de méthode `strtok` qui :

lorsqu'elle est appelée avec un chaîne de caractère (`CharSequence`) et un délimiteur (`char`), renvoie un `CharSequence` correspondant à la chaine du début de la chaine jusqu'à la prochaine occurrence du délimiteur.

lorsqu'elle est appelée avec `null` et un délimiteur, renvoie un `CharSequence` de la position lors du dernier appel à la méthode jusqu'à la prochaine occurrence du délimiteur.

Voici le code :

```
public static CharSequence strtok(CharSequence input,char
delimiter) {
    int offset;
    if (input==null) {
        input=lastInput;
        if (input==null)
            return null;

        offset=lastOffset;
    }
    else {
```

```

        lastInput=input;
        offset=0;
    }

    for(int i=offset;i<input.length();i++) {
        if (input.charAt(i)==delimiter) {
            lastOffset=i+1;
            return input.subSequence(offset,i);
        }
    }
    lastInput=null;
    return input.subSequence(offset, input.length());
}

private static CharSequence lastInput;
private static int lastOffset;

```

et un exemple :

```

CharSequence seq1=strtok("toto est beau", ' '); // toto
CharSequence seq2=strtok(null, ' '); // est
CharSequence seq3=strtok(null, ' '); // beau
CharSequence seq4=strtok(null, ' '); // null

```

- 1 Indiquer quel est le problème si deux threads différentes exécute en même temps un appel à strtok.
- 2 Faire en sorte que la méthode strtok soit thread-safe en utilisant des variables locales à un thread.

Exercice 3 - Liste concurrente et atomicité

On souhaite écrire une implantation de liste chaînée concurrente n'utilisant ni section critique ni verrou.

La liste devra posséder deux méthodes addLast(E e) et toString() permettant respectivement d'ajouter un élément en queue de liste et d'afficher la liste. Lors de la création, la liste sera créée avec un maillon.

L'idée pour éviter toutes synchronisations lors de l'ajout consiste à récupérer le maillon à la fin de la liste puis à tester en une opération atomique si le maillon est toujours la fin de la liste et si oui à lui assigner le nouveau maillon créé. Si ce n'est pas le dernier maillon, on récupère de nouveau la fin de la liste et l'on recommence le même algorithme.

Dans un premier temps, nous allons définir la liste comme contenant juste un élément ainsi qu'une référence sur un autre élément de la liste. L'ajout à la fin de la liste se fera par un parcours du chaînage.

- 1 Définir la classe ConcurrentLink avec ces champs element et next ainsi que son constructeur ConcurrentLink(E firstElement).
- 2 Expliquer pourquoi le champ element doit être déclaré final ?
- 3 Implanter la méthode addLast en utilisant pour tester et assigner en une opération atomique, la classe AtomicReferenceFieldUpdater du paquetage java.util.concurrent.atomic.
- 4 Implanter la méthode toString.

On souhaite maintenant séparer l'implantation de la liste de celle d'un maillon dans le but de stocker le dernier maillon pour effectuer un ajout plus efficace.

- 1 Définir la classe ConcurrentList pour quel contiennent deux champs head et tail correspondant respectivement au début et à la fin de la liste ainsi que la classe interne

Entry correspondant à un maillon de la liste chaînée.

Pour simplifier les choses, la liste ne pourra être vide, elle sera donc construite avec un élément.

2 Planter la méthode `addLast`

Attention, il y a un petit piège lorsque l'on met à jour la référence correspondant à la fin de la liste.

3 Planter la méthode `toString`.

Si vous avez réussi bravo, vous venez de ré-planter une partie de la classe `ConcurrentLinkedList`.