TP6 JAVA

Concepts abordés

- Threads, programmation concurrente

Consignes générales de travail

- Commencez chaque TP dans un nouveau *Projet Java (ici TP4)*
- Créez toujours un (ou plusieurs) paquetages pour contenir ses classes (ne jamais utiliser le paquetage par défaut).
- Écrivez chaque nouvelle classe dans son propre fichier (portant le nom de la classe).
- Pour ce TP, déclarez systématiquement vos attributs en private et vos méthodes en *package friendly* ou public.
- Définissez systématiquement les méthodes equals():boolean, hashcode:int
  et toString():String *pour chaque classe* que vous écrivez.
- Lisez chaque exercice en entier avant de commencer vos réponses.

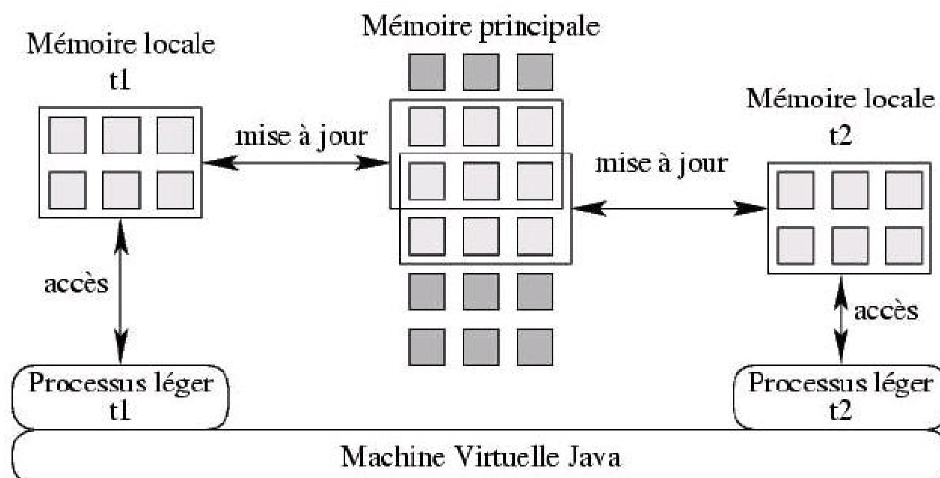
Pour mener à bien ces exercices, il est impératif d'ouvrir

1) **la Javadoc** **http://docs.oracle.com/javase/6/docs/api/**
2) le tutoriel de Sun sur le collections
   http://docs.oracle.com/javase/tutorial/collections/index.html

L'3 TP JAVA 2015-2016

Les problèmes de la concurrence

- Les instructions des différentes threads peuvent, a priori, être exécutées dans n'importe quel ordre.
- Un même espace mémoire central (JVM) partagé par toutes les threads, mais chacune dispose de ses propres registres
- Chaque thread dispose d'une mémoire locale (registres) pour stocker les « copies de travail » des variables
- Cela réduit le temps d'accès mais crée « plusieurs versions » de la même variable
- Cela pose donc des problèmes de cohérence et de mise à jour
- Chaque processus léger stocke les valeurs qu'il utilise dans ses registres: la mise à jour n'est pas systématique
- Le mot clé volatile, utilisé comme un modificateur de champ, assure la cohérence entre la mémoire de travail et la mémoire principale (pas de cache possible); la mise à jour est forcée pour prendre en compte les dernières modifications.
- un verrou sur l'objet class est posé pour tout accès à une méthode statique
- le verrou est posé sur this si l'appel se fait sur une méthode non statique
- un verrou sur this est a proscrire car this est accessible de l'extérieur et peut donc être verouillé par un tiers. Il faut gérer en interne les verrous avec un moniteur privé.



L'3 TP JAVA 2015-2016

ATTENTION :
Il n'y a pas vraiment de mémoires principale et locales. En fait, mémoire locale réfère au bancs de registres et mémoire principale à la RAM.
En C, le mot clé register permet de forcer le stockage d'une variable dans les registres. En java, c'est le comportement par défaut. C'est pour empécher ce comportement par défaut que l'on utilise volatile.

Exercice 1

On souhaite créer 4 threads (le nombre peut changer) qui exécutent un même code affichant les nombres de 0 à 5 000.
Histoire de différencier les threads à l'affichage, chaque thread affichera en plus du nombre courant un numéro (0 pour la première thread, 1 pour la seconde, etc).
Par exemple, on pourra obtenir ce type d'affichage :

```
...
hello 0 1714
hello 0 1715
hello 0 1716
hello 0 1717
hello 1 1096
hello 1 1097
hello 1 1098
hello 1 1099
 ...
```

1) Rappeler à quoi sert un objet de type ***Runnable***?

2) Ecrire dans un premier temps, une ***classe HelloThread*** qui crée et démarre 4 threads qui affichent les nombres de 0 à 5 000 (sans numéro unique par thread donc).

3) Exécutez le programme plusieurs fois, que remarque t'on ?
Puis, en regardant l'affichage (scroller au besoin), qu'y-a-t'il de bizarre ?
Est-ce que tout ceci est bien normal ?

4) Modifiez votre code pour afficher en plus le numéro de chaque thread.

Exercice 2

On souhaite afficher le message "le programme est fini", lorsque tous les threads ont fini de faire leurs calculs.

1) Recopiez le programme de l'exercice précédent dans une nouvelle ***classe HelloThreadJoin*** puis modifiez le pour que soit affiché le message "le programme est fini" lorsque **tous** les threads ont fini leurs calculs.
Si vous cherchez comment attendre que des threads aient fini d'exécuter leur Runnable, la méthode que vous cherchez est **Thread.join.**

```
Thread t = new Thread(this);
t.start();
t.join();
System.out.println("La thread t a fini son Runnable");
```

Exercice 3

On souhaite modifier le programme précédent pour qu'au lieu d'afficher les nombres, on les stocke dans une unique ***ArrayList<Integer>*** (une seule liste pour toutes les threads) dont on affichera la taille à la fin du programme.

1) Recopiez la ***classe HelloThreadJoin*** dans une nouvelle ***classe HelloListBug*** puis modifiez la pour ajouter les nombres au lieu de les afficher et pour afficher la taille finale une fois toutes les threads terminés.
Exécuter le programme plusieurs fois et noter les différents affichages (oui, même les exceptions).
2) Expliquer quel est le problème lorsqu'une exception est levée. Pour comprendre, il faut regarder le code de la ***méthode ArrayList.add***.

```
/**
   * Appends the specified element to the end of this list.
   *
   * @param e element to be appended to this list
   * @return <tt>true</tt> (as specified by {@link java.util.Collection#add})
   */
  public boolean add(E e) {
     ensureCapacityInternal(size + 1);  // Increments modCount!!
     elementData[size++] = e;
     return true;
  }

  private void ensureCapacityInternal(int minCapacity) {
     if (elementData == EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
     }
     ensureExplicitCapacity(minCapacity);
  }

  private void ensureExplicitCapacity(int minCapacity) {
     modCount++;

     // overflow-conscious code
     if (minCapacity - elementData.length > 0)
        grow(minCapacity);
  }
```

3) Puisque l'exception se produit lorsque l'on agrandit l'***ArrayList***, on peut essayer de la créer avec la bonne taille.

```
ArrayList<Interger> list = new ArrayList(5_000*4);
```

Exécuter le programme plusieurs fois et noter les différents affichages.

Expliquer quel est le problème.

4) Corriger le problème et vérifier que la correction que vous avez effectuée, exécute bien les threads en parallèle et non pas les unes derrière les autres.

Exercice 4

On cherche à savoir combien d'itération d'une boucle on peut faire en 100 millisecondes, le code suivant est donné :

```
package concurrent;

public class CounterBug implements Runnable {
	private boolean stop;

	public void run() {
		int localCounter = 0;
		for(;;) {
			if (stop) {
				break;
			}
			localCounter++;
		}
		System.out.println(localCounter);
	}

	public void stop() {
		stop = true;
	}

	public static void main(String[] args) throws InterruptedException {
		CounterBug counter = new CounterBug ();
		Thread thread = new Thread(counter);
		thread.start();
		Thread.sleep(100);
		counter.stop();
		thread.join();
	}
}
```

1) Sans exécuter le code, que fait ce programme ?

2) Vérifier en exécutant le programme (plusieurs fois) si vous avez vu juste.

3) Comment doit on corriger le problème ?

Modifier la ***classe CounterBug*** en conséquence.

4) On cherche maintenant a accelérer le code de ***CounterBug*** en utilisant le mot clé ***volatile*** au lieu des blocs ***synchronized.***

Créer une classe ***CounterVolatile*** qui n'utilise pas de block synchronized.

Comment appelle t-on les implantations qui n'ont ni blocs synchronized ni lock ?