TRAINA Tanguy
INFO 2 - Logiciel

Collections concurrentes

TP1 - Volatile, Opérations atomiques et CompareAndSet

Blocs synchronized

- **Problème :**
  - Pour optimiser, la VM fait des copies en registres des valeurs utilisées par certaines méthodes
  - Cela rend les méthodes non *thread-safe*
- **Solution**
  - On utilise des blocs *synchronized :*
    - forcent les lectures/écritures des champs d'un objet en RAM plutôt que dans des registres
    - garantissent qu'un seul thread à la fois puisse accéder au bloc en question

Champs *volatile*

- Autre solution pour gérer les problèmes de lectures/écritures en RAM

  Mot-clé utilisé sur un champs d'un objet

- La JVM garantit que la valeur lue est toujours à jour

- Déclaration :

```
private volatile boolean stop;
```

Atomicité

- **Opération atomique:**
  - désigne une opération effectuée atomiquement par le processeur
    - Un *thread* ne peut pas être *déschedulé* au cours d'une opération atomique
      - Remarque : l'opération " ++ " n'est pas atomique

Atomicité

- **Classe *AtomicReference* :**
  - *compareAndSet (E oldValue, E newValue)* :
    - Compare la valeur trouvée en mémoire avec celle fournie en paramètre
    - Si les deux valeurs diffèrent, renvoit faux
    - Sinon, modifie la valeur, et renvoit vrai

    → Permet de savoir que la valeur a été modifiée (par un autre thread), et donc de la relire avant d'agir

Atomicité

- **Classe *AtomicReference* :**
  - Sacrifice de l’efficacité au profit de la comptabilité
  - Utilisation d’un champs *volatile* pour représenter la référence
    - Coûte cher en accès mémoire

Atomicité

- **Classe AtomicInteger:**
  - *compareAndSet (int oldValue, int newValue)* :
    - Fonctionnement similaire à celui détaillé précédemment
  - *getAndIncrement()* :
    - Permet une incrémentation atomique selon l'architecture du processeur
    - Renvoit la valeur précédente

Lock-free

- **Implémentation lock-free :**
  - Le code ne contient ni bloc *synchronized,* ni *locks*