Java Avancé

Thread

Emerite NEOU

Thread

Partage de Variable

Thread

Partage de Variable

Probleme

```
public static void main(String[] args) {
    int a = 1 ;
    int b = 2 ;
    int c = a + b ;
}
```

Probleme

```
public static void main(String[] args) {
    int a = 1 ;
    int b = 2 ;
    int c = a + b ;
}
```

```
public static void main(java.lang.String[]) ;
  descriptor : ([Ljava/lang/String ;)V
  flags : (0x0009) ACC_PUBLIC, ACC_STATIC
  Code :
    stack=2, locals=4, args_size=1

    0 : iconst_1
    1 : istore_1

    2 : iconst_2
    3 : istore_2

    4 : iload_1
    5 : iload_2
    6 : iadd
    7 : istore_3

    8 : return
    ...
```

Probleme

```
public static void main(String[] args) {
    int a = 1 ;
    int b = 2 ;
    int c = a + b ;
}
```

Probleme

```
public static void main(String[] args) {
    int a = 1 ;
    int b = 2 ;
    int c = a + b ;
}
```

```
public static void main(java.lang.String[]) ;
  descriptor : ([Ljava/lang/String ;)V
  flags : (0x0009) ACC_PUBLIC, ACC_STATIC
  Code :
    stack=2, locals=4, args_size=1

    0 : iconst_1
    1 : istore_1

    2 : iconst_2
    3 : istore_2

    4 : iload_1
    5 : iload_2
    6 : iadd
    7 : istore_3

    8 : return
    ...
```

Probleme

```
public class Counter {
  private int value ;
  public void add10000() {
    for(int i = 0 ; i < 10000 ; i++) {
      value++ ;
    }
  }
  public static void main(String[] args) throws InterruptedException {
    Counter counter = new Counter() ;
    Runnable runnable = new Runnable() {
      @Override public void run() {
        counter.add10000() ;
      }
    } ;
    Thread t1 = new Thread(runnable) ;
    Thread t2 = new Thread(runnable) ;
    t1.start() ; t2.start() ;
    t1.join() ; t2.join() ;
    System.out.println(counter.value) ;
  }
}
```

- Affiche ?

Probleme

- Affiche ?
  - 14213 (? ?)
  - 20000 (ah !)
  - 12157 (? ? ?)

Probleme

- Affiche ?
  - 14213 (? ?)
  - 20000 (ah !)
  - 12157 (? ? ?)
- Pourquoi ?

Probleme

- Affiche ?
  - 14213 (? ?)
  - 20000 (ah !)
  - 12157 (? ? ?)
- Pourquoi ?
  - Les threads ne mettent pas à jour la memoire commune.

Probleme

- Affiche ?
  - 14213 (??)
  - 20000 (ah !)
  - 12157 (???)
- Pourquoi ?
  - Les threads ne mettent pas à jour la memoire commune.
  - `i++` n'est pas une opération **atomique**, mais une lecture, une incrémentation et une écriture

Solution

Solution ?

Solution

Solution ?

- Rendre les variables atomiques : pas toujours possible.

Solution

Solution ?

- Rendre les variables atomiques : pas toujours possible.
- Empêcher la lecture tant qu'écriture non terminée : **section critique**.

Et volatile ?

Oblige l'ecriture et la lecture dans la mémoire commune

Et volatile ?

Oblige l'ecriture et la lecture dans la mémoire commune $->$ n'empêche d'être scheduler

Atomic Variable

Des classes avec des methods qui ne peuvent pas être descheduler.

Atomic Variable

Des classes avec des methods qui ne peuvent pas être descheduler.

```
private AtomicInteger value = new AtomicInteger(0) ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    value.addAndGet(1) ;
  }
}
```

Section Critique

Une seule thread peut accéder à cette partie du code.

Section Critique

Une seule thread peut accéder à cette partie du code.

```
private int value ;
private final Object lock = new Object()  ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    synchronized (monitor) {
      value++ ;
    }
  }
}
```

Section Critique

- Si déjà utilisé, la thread est descheduler et retentera plus tard

Section Critique

- Si déjà utilisé, la thread est descheduler et retentera plus tard

Section Critique - Dead Lock

```
public class PingPong {
  private final Object monitor1 = new Object() ;
  private final Object monitor2 = new Object() ;

  public void ping() {
    synchronized (monitor1) {
      synchronized (monitor2) {
        //
      }
    }
  }
  public void pong() {
    synchronized (monitor2) {
      synchronized (monitor1) {
        //
      }
    }
  }
}
```

Section Critique - best practices

- Utiliser des variables dediées,

Section Critique - best practices

- Utiliser des variables dediées,
- finale, private et sans getter,

Section Critique - best practices

- Utiliser des variables dediées,
- finale, private et sans getter,
- section les plus courtes possibles.

Section Critique - best practices

```
private List<Integer> values ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    synchronized (value) {
      values.add(1) ;
    }
  }
}
```

NON !

Section Critique - best practices

```
private List<Integer> values ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    synchronized (value) {
      values.add(1) ;
    }
  }
}
```

NON !

```
private List<Integer> values ;
private final Object lock = new Object()  ;
public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    synchronized (lock) {
      values.add(1) ;
    }
  }
}
```

OUI !

Section Critique - best practices

```
private final Object lock = new Object()  ;
public boolean exist(String name, int num) {
  synchornised(lock) {
    String id = name+num ;
    return l.contains(id) ;
  }
}
```

NON !

Section Critique - best practices

```
private final Object lock = new Object()  ;
public boolean exist(String name, int num) {
  synchornised(lock) {
    String id = name+num ;
    return l.contains(id) ;
  }
}
```

NON !

```
private final Object lock = new Object()  ;
public boolean exist(String name, int num) {
  String id = name+num ;
  synchornised(lock) {
    return l.contains(id) ;
  }
}
```

OUI !

Attendre une modification

On souhaite attendre qu'une autre thread modifie une variable

- attendre que l'autre thread se finisse.

Attendre une modification

On souhaite attendre qu'une autre thread modifie une variable

- attendre que l'autre thread se finisse.
- avec la method wait

Attendre une modification

```
private Connection c = null ;
private final Object lockInit = new Object()  ;



public void initConf() {
  synchronised(lockInit) {
    if(c == null) lockInit.wait() ;
  }
  ...
}
```

Attendre une modification

```
private Connection c = null ;
private final Object lockInit = new Object()  ;



public void initConf() {
  synchronised(lockInit) {
    if(c == null) lockInit.wait() ;
  }
  ...
}
```

probleme : et si la thread se reveille ?

Attendre une modification

```
private Connection c = null ;
private final Object lockInit = new Object()  ;



public void initConf() {
  synchonised(lockInit) {
    while(c == null) lockInit.wait() ;
  }
  ...
}
```

Signaler un évènement attendu par un wait

On dire à une autre thread qu'un évènement attendu est arrivé :

```
private Connection c = null ;
private final Object lockInit = new Object()  ;



public void createConnection() {
  synchronised(lockInit) {
    c = new Connection() ;
    lockInit.notify() ;
  }
  ...
}
```

Locks

Meme principe que synchornised mais avec des outils en plus.

Locks

Meme principe que synchornised mais avec des outils en plus.

Locks

```
private int value ;
 private final Lock lock = new ReentrantLock()  ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    lock.lock() ;
    value++ ;
    lock.unlock() ;
  }
}
```

Locks

```
private int value ;
 private final Lock lock = new ReentrantLock()  ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    lock.lock() ;
    value++ ;
    lock.unlock() ;
  }
}
```

JAMAIS

Locks

```
private int value ;
 private final Lock lock = new ReentrantLock()  ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    lock.lock() ;
    value++ ;
    lock.unlock() ;
  }
}
```

JAMAIS

```
private int value ;
 private final Lock lock = new ReentrantLock()  ;

public void add10000() {
  for(int i = 0 ; i < 10000 ; i++) {
    lock.lock() ;
    try {
      value++ ;
    }  finally {
      lock.unlock() ;
    }
  }
}
```

Locks

Meilleur que synchornised car :

- On peut prendre un lock dans une method et la rendre dans une autre ...

Locks

Meilleur que synchornised car :

- On peut prendre un lock dans une method et la rendre dans une autre ...
- fair : si plusieurs threads en attente sur le verrou, favorise celle qui attend depuis plus longtemps (sinon hasard mais plus rapide).

Locks

Meilleur que synchornised car :

- On peut prendre un lock dans une method et la rendre dans une autre ...
- fair : si plusieurs threads en attente sur le verrou, favorise celle qui attend depuis plus longtemps (sinon hasard mais plus rapide).
- Un controle plus fin.