Concurrence memory model

Rémi Forax

WORA !

- Java permet à un même code de s'exécuter sur plusieurs architecture/OS différentes
  - Write Once Run Anywhere (WORA)
- Problème, chaque Architecture/OS à ses propres rêgles de fonctionnement différentes les unes des autres
  - Comment guarantir une même exécution ?

Modèle de mémoire

S'abstrait d'une machine particulière et fourni des garanties sur l'ordre d'exécution

- Une opération voie l'état d'une opération précédente

  “happen before”

- Si rien n'est spécifié par le modèle de mémoire
  - Aucune garantie

Pour une même thread

La lecture de la valeur d'un même champ d'une même instance ou une même variable locale voit toujours la dernière écriture

```
Point p = ...
p.x = 1;
p.x = 3;
int a = p.x;        // <==> a = 3
```

ce n'est valable que pour une même thread !

Entre plusieurs threads

Il n'y a aucune garantie entre des threads différentes

Par exemple,

avec p un Point visible par les deux threads

Thread 1	Thread 2
p.x = 1	p.x = 3
p.x ? // 1 ou 3	p.x ? // 1 ou 3

Il faut utiliser des constructions spécifiques (final, volatile, synchronized, etc)

start() et join()

thread.start() guarantie que toute les écritures précédentes seront visibles par la thread qui démarre

```
Point p = new Point();
Runnable runnable = () -> {
  System.out.println(p);  // garantie que p est non null
};
Thread t = new Thread(runnable);
t.start();
```

start() et **join**()

thread.join() guarantie que les écritures faites dans la thread “jointe” seront visible à la thread courante

```
class MyRunnable implements Runnable {
  private int result;
  public void run() {
    result = 3;
  }
});
MyRunnable runnable = new MyRunnable();
Thread t = new Thread(runnable);
...
t.join();
System.out.println(runnable.result);   // 3
```

Block static

Les initialisations faites dans un bloc statique sont visibles par toutes les threads qui utilisent la classe

```
class A {
  private static Object o;
  static {
    o = new Object();
  }

  public static void main(String[] args) {
    new Thread(() -> {
      System.out.println(A.o); // ne peut pas être null
    }).start();
    System.out.println(A.o); // ne peut pas être null
  }
}
```

Beurk, car pas une constante !!

Publication

- Un champ déclaré final est visible par toutes les threads après la fin du constructeur !

```
class A {
  final int value;
  A(int value) {
    this.value = value;
  }
  private static A a;
  public static void main(String[] args) {
    new Thread(() -> {
      if (a != null) {
        System.out.println(a.value);  // 7
      }
    }).start();
    a = new A(7);
  }
}
```

Problème de publication (1)

Et si on oublie le final ??

- On peut voir la valeur par défaut ('\0', 0, 0.0, null)

```
class A {
  final int value;
  A(int value) {
    this.value = value;
  }
  private static A a;
  public static void main(String[] args) {
    new Thread(() -> {
      if (a != null) {
        System.out.println(a.value);  // 7 ou 0
      }
    }).start();
    a = new A(7);
  }
}
```

Problème de publication (2)

Et si on publie this avant la fin du constructeur ?

- On peut voir la valeur par défaut ('\0', 0, 0.0, null)

```
class A {
  final int value;
  A(int value) {
    this.value = value;
    A.a = this;                    // oh no !
  }
  private static A a;
  public static void main(String[] args) {
    new Thread(() -> {
      if (a != null) {
        System.out.println(a.value);  // 7 ou 0
      }
    }).start();
    new A(7);
  }
}
```

Volatile

3 effets !

- L'écriture dans un champ volatile rend visible toutes les écritures précédentes
- La lecture dans un champ volatile oblige les lectures suivantes a être rechargé à partir de la RAM
- L'écriture d'une valeur 64 bits (long ou double) est vu comme atomique même sur une machine 32 bits

Exemple de volatile

```
class A {
  int value;
  volatile boolean done;

  public void init(int value) {
    this.value = value;
    this.done = true;    // volatile write, donc value est écrit en RAM
  }
  public static void main(String[] args) {
    A a = new A();
    Thread t = new Thread(() -> {
      a.init(9);
    }).start();
    if (a.done) {              // volatile read, cache invalidation
      System.out.println(a.value);  // doit être rechargé de la RAM donc 9
    }
  }
}
```

Synchronized et lock

- L'entrée dans un bloc synchronized oblige la relecture des champs à partir de la RAM
- En sortie d'un bloc synchronized, toutes les écritures sont visibles en RAM
  - Elles peuvent être visible en RAM avant la sortie

Exemple de synchronized

- class A {
```
  volatile int value;
  volatile boolean done;                // les deux champs n'ont pas besoin d'être déclaré volatile !
  final Object lock = new Object();   // si pas final problème de publication !

  public void init(int value) {
    synchronized(lock) {
      this.value = value;
      this.done = true;
    }   // value et true sont écrit en RAM
  }
  public static void main(String[] args) {
    A a = new A();
    Thread t = new Thread(() -> {
      a.init(9);
    }).start();
    synchronized(a.lock) {
      if (a.done) {              // done et value sont rechargés à partir de la RAM
        System.out.println(a.value);  // 9
      }
    }
  }
}
```

Exemple avec le singleton

Design pattern qui garantie qu'il existe 1 seul instance d'une classe

```
public class DB {
  // plein de déclaration de champs, c'est un gros objet

  private DB() {
    // init des champs
  }

  private static final DB = new DB();
  public static DB getSingleton() {
    return INSTANCE;
  }
}
```

Marche très bien car les classes sont chargés de façon paresseuse en Java (et pas en C++)

Singleton lazy ?

On essaye de transposé un problème de C++ que Java n'a pas

```
public class DB {
  // plein de déclaration de champs, c'est un gros objet
  private DB() {
    // init des champs
  }

  private static ~~final~~ DB INSTANCE;
  public static DB getSingleton() {
    if (INSTANCE == null) {
      INSTANCE = new DB();
    }
    return INSTANCE;
  }
}
```

Le code est **pas thread safe** !

Singleton lazy ?

On peut résoudre le problème avec un synchronized !

```
public class DB {
  // plein de déclaration de champs, c'est un gros objet
  private DB() { /* init des champs */ }

  private static ~~final~~ DB INSTANCE;
  private static final Object lock = new Object();
  public static DB getSingleton() {
    synchronized(lock) {
      if (INSTANCE == null) {
        INSTANCE = new DB();
      }
      return INSTANCE;
    }
  }
}
```

Pas de problème ce code marche !

Et si on veut éviter le synchronized ?

```
public class DB {
  // plein de déclaration de champs, c'est un gros objet
  private DB() { /* init des champs */ }

  private static ~~final~~ DB INSTANCE;
  private static final Object lock = new Object();
  public static DB getSingleton() {
    if (INSTANCE == null) {
      synchronized(lock) {
        INSTANCE = new DB();
        return INSTANCE;
      }
    }
    return INSTANCE;
}
```

Ce code **ne marche pas** !

Double check locking

- Design pattern qui ne marche pas :(

```java
public class DB {
  // plein de déclaration de champs, c'est un gros objet
  private DB() { /* init des champs */ }

  private static final DB INSTANCE;
  private static final Object lock = new Object();
  public static DB getSingleton() {
    if (INSTANCE == null) {
      synchronized(lock) {
        if (INSTANCE == null) {
          INSTANCE = new DB();
        }
        return INSTANCE;
      }
    }
    return INSTANCE;
}
```

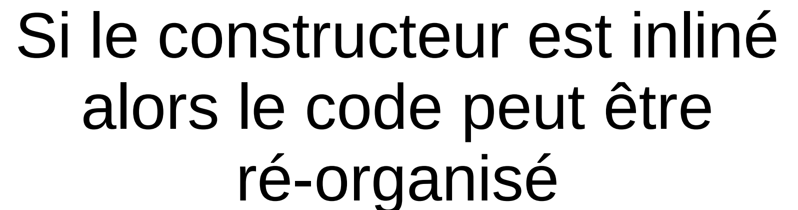
Et non, ça marche pas !

Double check locking (2)

- On peut publier l'objet pas fini d'être initialisé !

```
public class DB {
  // plein de déclaration de champs, c'est un gros objet
  private DB() { /* init des champs */ }

  private static final DB INSTANCE;
  private static final Object lock = new Object();
  public static DB getSingleton() {
    if (INSTANCE == null) {
      synchronized(lock) {
        if (INSTANCE == null) {
          DB tmp = new DB
          tmp.field1 = ...; tmp.field2 = ...
          INSTANCE = tmp;
        }
        return INSTANCE;
      }
    }
    return INSTANCE;
}
```



Double check locking (3)

On peut publier l'objet pas fini d'être initialisé !

```
public class DB {
  // plein de déclaration de champs, c'est un gros objet
  private DB() { /* init des champs */ }

  private static final DB INSTANCE;
  private static final Object lock = new Object();
  public static DB getSingleton() {
    if (INSTANCE == null) {
      synchronized(lock) {
        if (INSTANCE == null) {
          DB tmp = new DB
          INSTANCE = tmp;
          tmp.field1 = ...; tmp.field2 = ...
        }
        return INSTANCE;
      }
    }
    return INSTANCE;
}
```

(Dans `private static final DB INSTANCE;`, le mot `final` est barré.)

Ré-organisation possible !

INSTANCE est initialisé avant les champs !

Double check locking (4)

On peut résoudre ce problème en marquant INSTANCE volatile,
mais dans ce cas, le code reste lent car on paye la lecture volatile !

```
public class DB {
  // plein de déclaration de champs, c'est un gros objet
  private DB() { /* init des champs */ }

  private static volatile DB INSTANCE;
  private static final Object lock = new Object();
  public static DB getSingleton() {
    if (INSTANCE == null) {      // volatile read
      synchronized(lock) {
        if (INSTANCE == null) {  // volatile read
          DB tmp = new DB
          tmp.field1 = ...; tmp.field2 = ...
          INSTANCE = tmp;        // volatile write
        }
        return INSTANCE;
      }
    }
    return INSTANCE;
}
```

Tous ça pour résoudre
un problème
qui n'existe pas en Java