Session: Concurrency and Parallelism in Java
.....…

0. Hardware und Compiler:
Stalking the Lost Write: Memory Visibility in Concurrent Java
Jeff Berkowitz, New Relic
December 2013
(im gleichen Verzeichnis).

Schauen Sie sich die sog. „Executions“ der beiden Threads nochmal an. Erkennen Sie, dass es nur die zufällige Ablaufreihenfolge der Threads ist, die die verschiedenen Ergebnisse erzeugt? Das sind die berüchtigten „race conditions“…

Tun Sie mir den Gefallen und lassen Sie das Programm Mycounter in den se2_examples selber ein paar Mal ablaufen. Tritt die Race Condition auf? Wenn Sie den Lost Update noch nicht verstehen: Bitte einfach nochmal fragen!

Tipp: Wir haben ja gesehen, dass i++ von zwei Threads gleichzeitig auseführt increments verliert (lost update). Wir haben eine Lösung gesehen in MyCounter: das Ganze in einer Methode verpacken und die Methode „synchronized“ machen. Also die „Ampellösung“ mit locks dahinter. Das ist teuer und macht unseren Gewinn durch mehr Threads kaputt.

Hm, könnte man i++ ATOMAR updaten ohne Locks??
Suchen Sie mal nach „AtomicInteger“ im concurrent package von Java!

1. Arbeitet Ihr Gehirn concurrent oder parallel (:-)?

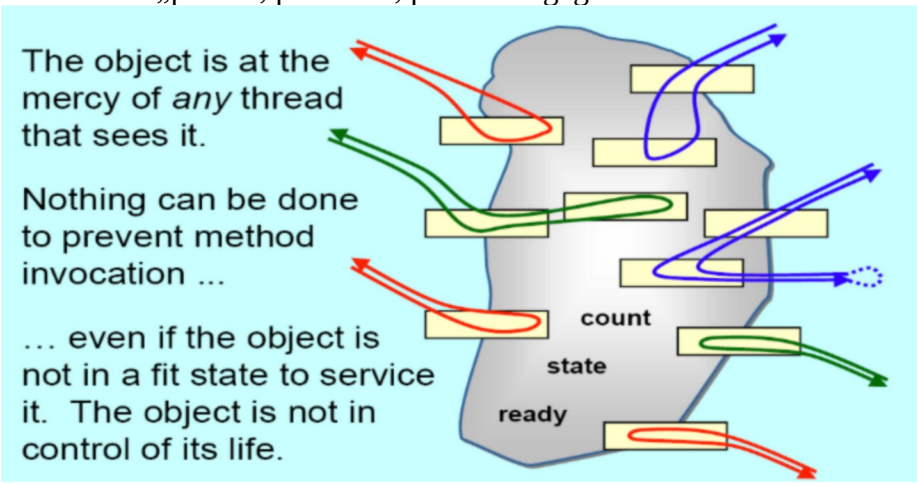
This is a rather philosophical question. To answer it we have to know the difference between concurrency and parallelism:

Concurrency is when something does multiple tasks at the same time.
Parallelism on the other hand is when something splits a single task into multiple units and does them simultaneously.

The example of calculation:

We do not split up calculations into smaller units and do them parallely. However, some might argue that we are able to do something while calculating (e.g. eating, which does not require that much brain). So in conclusion, we might be able to do things concurrently but not parallely.

2. Multi-threading hell: helfen „private, protected, public“ dagegen?



Ist Ihnen klar warum die OO-Kapselung durch „private“ Felder nicht hilft bei Multithreading?

Two threads could write to the same variable through public (setter) methods. So, making a variable private won’t help about concurrent modification issues (e.g. lost update).

3. Muss der schreibende Zugriff von mehreren Threads auf folgende Datentypen synchronisiert werden?
- statische Variablen
- Felder/Attribute in Objekten
- Stackvariablen
- Heapvariablen (mit new() allokierte..)

Static variables: Yes, access has to be synchronized. Static variables are shared resources and thus may be suspect to lost updates etc.

Fields in objects: Yes, access has to be synchronized as long as the object is used by multiple threads. If not, it does not matter.

Stack variables: No, access does not have to be synchronized as stack variables are not shared.

Heap variables: Yes, access does have to be synchronized.

4. Machen Sie die Klasse threadsafe (sicher bei gleichzeitigem Aufruf). Gibt‘s Probleme? Schauen Sie ganz besonders genau auf die letzte Methode getList() hin.

```
class Foo {
                private ArrayList aL = new ArrayList();
                public int elements;
                private String firstName;
                private String lastName;

                public Foo () {};

                public void synchronized setfirstName(String fname) {
                        firstName = fname;
                        elements++;
                }
                public setlastName(String lname) {
                        lastName = lname;
                        elements++;
                }
                public synchronized ArrayList getList () {
                return aL;
        }
  }
```

```java
public class Foo {
    private final ArrayList<Object> aL = new ArrayList<>();
    private int elements;
    private String firstName;
    private String lastName;

    public Foo () {};

    public synchronized void setfirstName(String fname) {
        firstName = fname;
        elements++;
    }
    public synchronized void setlastName(String lname) {
        lastName = lname;
```

```
        elements++;
    }
    public synchronized List<Object> getList () {
        return Collections.unmodifiableList(aL);
    }
}
```

5. kill_thread(thread_id) wurde verboten: Wieso ist das Töten eines Threads problematisch? Wie geht es richtig?

Killing or stopping threads is depracated because stopping a thread may leave corrupted objects behind. See: https://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

6. Sie lassen eine Applikation auf einer Single-Core Maschine laufen und anschliessend auf einer Multi-Core Maschine. Wo läuft sie schneller?

This depends on whether the application makes use of multiple threads. If not, the difference is (if the single-core performane of both machines is the same) neglectable.

7. Was verursacht Non-determinismus bei multithreaded Programmen? (Sie wollen in einem Thread auf ein Konto 100 Euro einzahlen und in einem zweiten Thread den Kontostand verzehnfachen)

When two threads access data simoultaneously, weird things can happen. Example:

int money = 0;

1. Thread 1 reads the value.
2. Thread 2 adds 100 to the value.
3. Thread 1 multiplies the read value by 10.
4. Thread 1 writes the new value to the variable.

Result: money = 0

8. Welches Problem könnte auftreten wenn ein Thread produce() und einer consume() aufruft? Wie sähen Lösungsmöglichkeiten aus? Wenn es mehr als einen Producer und/oder Consumer gibt? Wo könnte/sollte „volatile“ hin? Wo eventuell „synchronized“?

```
1  public class MyQueue {
2
3       private Object store;
4       int flag = false; // empty

5       public  void produce(Object in) {
6                   while (flag == true)   ;  //full
7                   store = in;
                    flag = true; //full
9       }
10        public  Object consume() {
                    Object ret;
11                  while (flag == false)    ; //empty
12                  ret = store;
13                  flag = false;   //empty
14                  return ret;
15        }
16 }
```

The while-loops represent some sort of makeshift lock. Thus, they could be replaces altogether by the synchronized keyword. Then, you could use if(flag)... to check whether the queue is empty or not.

In any case flag should be declared as volatile so that the threads always read it from memory and not from cache.

9. Fun with threads.
    1. Was passiert mit dem Programm?
    2. Was kann auf der Konsole stehen?

```
public class C1 {
   public synchronized void doX (C2 c2) {
     c2.doX();
   }
   public synchronized void doY () {
     System.out.println("Doing Y");
   }

   public static void  main (String[] args)  {
   C1 c1 = new C1();
   C2 c2 = new C2();
   Thread t1 = new Thread(() -> { while (true) c1.doX(c2);});
   Thread t2 = new Thread(() -> { while (true) c2.doY(c1);});
   t1.start();
   t2.start();
   }
 };

  public class C2 {

   public synchronized void doX () {
     System.out.println("Doing X");
   }
   public synchronized void doY ( C1 c1) {
    c1.doY();
  }
```

1. Only c2.doX() will be executed as the methods all are synchronized and therefore have locks protecting them from multiple threads accessing them at the same time. As in this example both threads want to access synchronized methods from a single objects – which only provides one intrinsic lock - we run into a deadlock scenario.

2. The output will be a series of „Doing X“ as t1 will run just fine until t2 wants to access the lock in just the right moment.