Practical Concurrent and Parallel Programming

Kasper Østerbye

IT University of Copenhagen

Friday 2020-08-28

Plan for today

Meta - about how this course is organized

(See slides from yesterday for more)

Topics

- Java threads
- Java locking, the synchronized keyword
- Threads for performance

The teachers

- Course responsible: Kasper Østerbye
- PhD 1989, Aalborg University, programmed my first machine code in 1978 (wrote an assembler)
- Research interests: Programming languages and software understanding
  - Like programming languages: Java, C#, Kotlin, JavaScript, Lisp, Python, C++, Smalltalk, Forth, Icon, Prolog, Simula, Pascal, Concurrent Pascal, Beta, Eiffel, … (excuse me for mixing syntaxes once in a while).
- Joined ITU in 2000 (but has been away for 5 years)
- Co-teacher: Jørgen Staunstrup (will introduce himself later in the course)
- Teaching Assistants: Amund Ranheim Lome, Frederik Haagensen, and Holger Stadel Borum

Why this course?

Parallel programming is necessary

- The real world is parallel
  - Think of the atrium lifts: lifts move, buttons are pressed
  - Think of handling a million online banking customers
- For performance
- To share ressources (think virtual machines, sharing a printer, sharing a harddisk)

It is easy, and disastrous, to get it wrong

- Testing is even harder than for sequential code
- You should learn how to make correct parallel code
- in a real language, used in practice
- You should learn how to make fast parallel code
- and measure whether one solution is faster than another
- and understand why

Course contents

- Threads, locks, mutual exclusion, scalability
- Java 8 streams, functional programming
- Performance measurements
- Tasks, the Java executor framework
- Safety, liveness, deadlocks
- Testing concurrent programs
- Some more advanced concepts from Java and other languages (GO or Kotlin)

Learning objectives

See formal page in LearnIT

Expected prerequisites

- From the ITU course base:

  > Students must know the Java programming language very well, including inner classes and a first exposure to threads and locks, and event-based GUIs as in Swing or AWT.”

- Today we will briefly review the basics of Java threads

- Java synchronized methods and statements - Java’s final keyword
- Java inner classes and lambdas

Standard weekly plan

- Lectures Fridays in zoom
- feedback on Thursdays (zoom)
- Exercise hand-in: Thursday morning by 7
- On LearnIT, as a link to your itu github with solutions

See the google sheet linked from this weeks readme file for timeslots

Standard weekly plan

- Lectures Fridays in zoom
- feedback on Thursdays (zoom)
- Exercise hand-in: Thursday morning by 7
- On LearnIT, as a link to your itu github with solutions

See the google sheet linked from this weeks readme file for timeslots

Help during before feedback

- Group and team members - email in the sheet
- We need a vote - discussion on LearnIT forum or as github issues?

Course information online

See recording from yesterday

- Course LearnIT page, restricted access: https://learnit.itu.dk/course/view.php?id=3017108
- Course gitrepo, public access: https://github.itu.dk/kasper/PCPP-Public

- Overview of lectures and exercises
- Lecture slides and exercise sheets
- Example code
- List of all mandatory reading materials

Exercises

See recording from yesterday

- There are 13 sets of weekly exercises
- You are expected to work in groups of 2-3 students, and in teams of two groups.
- Hand in the solutions through LearnIT
- Each team will be given oral feedback on Thursday - a schedule is available.
- Hand-ins: ≥6 must be submitted, ≥5 approved - otherwise you cannot take the course examination
  - failing to get 5 approved costs an exam attempt (!!)
- Exercise may be approved even if not fully solved - It is possible to resubmit
  - Make your best effort: two serious attempts=one solved
- What is important is that you learn

Exam

- A 30 hour take-home written exam/project - Electronic submission in LearnIT
  - Followed by random sample “cheat check”
- Expected exam workload is 16 hours - Individual exam, no collaboration
  - All materials, including Internet, allowed - Always credit the sources you use
  - Plagiarism is forbidden - as always
- The old exams are on the public homepage (or will be as we get closer to exam)

Stuff you need

See recording from yesterday

- Buy Goetz et al: Java Concurrency in Practice - From 2006, still the best on Java concurrency
  - Most contents is relevant for C#/.NET too
- Free lecture notes and papers, see homepage
- A few other book chapters, see LearnIT
- Java 8 SDK installed on your computer - Java 7 or earlier will not work, later should be OK

What about other languages?

- .NET and C# are very similar to Java
- We will point out differences on the way
- Kotlin, Scala, F#, … build on JVM or .NET
- C and C++ have some differences
- GO, Rust, Python, …

And now to concurrency and parallelism

Threads and concurrency in Java

- A thread is
  - a sequential activity executing Java code
  - running at the same time as other activities
- Threads communicate via fields
  - That is, by updating shared mutable state

A thread-safe class for counting

```
class LongCounter {
  private long count = 0;
  public synchronized void increment() {
    count = count + 1;
  }
  public synchronized long get() {
    return count;
  }
} // TestLongCounter.java
```

- The state (field count) is private
- Only synchronized methods read and write it

A thread created from a Runnable

The thread's behavior is in the run method

```
final LongCounter lc = new LongCounter();
Thread t =
  new Thread(
      new Runnable() { //An anonymous inner class, and an instance of it
        public void run() {
          while (true)
                    lc.increment();
                    // When started, the thread will increment forever
            }
        }
); // TestLongCounter7.java
```

This only creates the thread, does not start it

Starting the thread

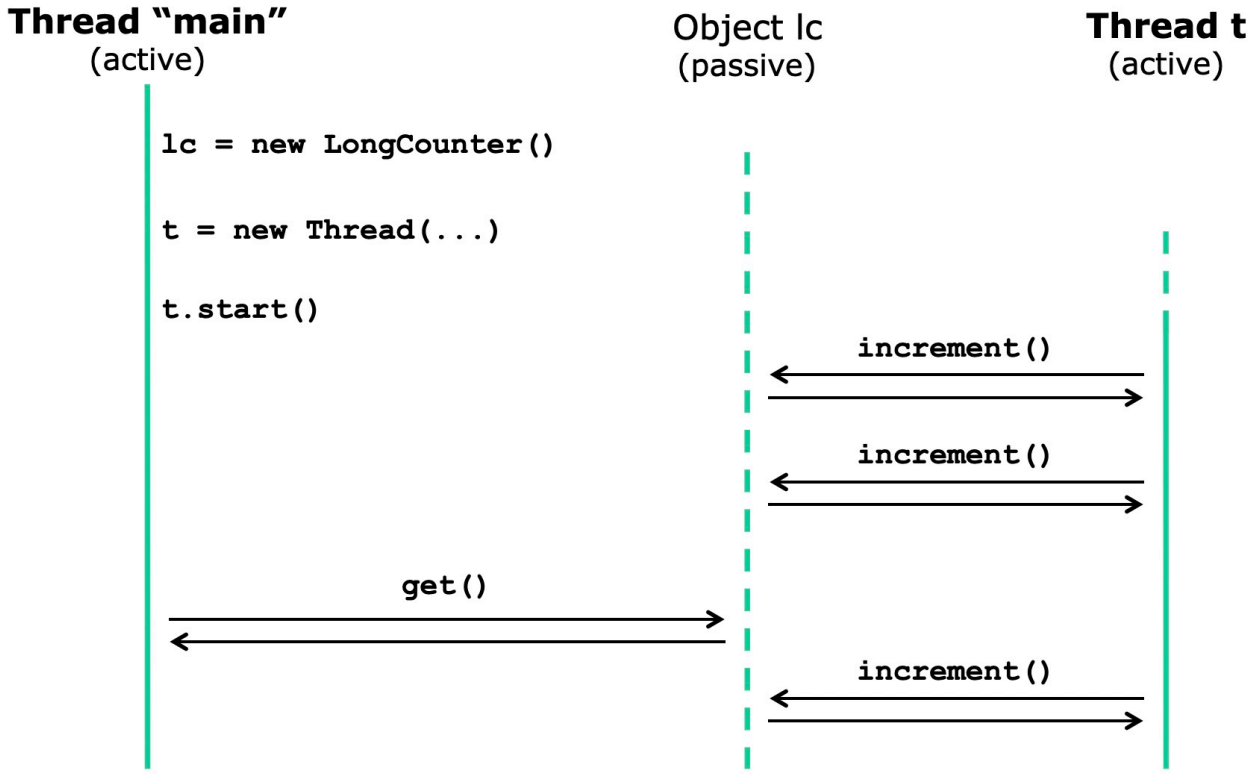
in parallel with the main thread

```
public static void main(String[] args) ... {
    final LongCounter lc = new LongCounter();
    Thread t = new Thread(new Runnable() { ... }); // previous slide
    t.start();
    System.out.println("Press Enter ... ");
    while (true) {
        System.in.read();
        System.out.println(lc.get());
    }
}
```

```
Press Enter to get the current value:
60853639
103606384
263682708
...
```

Creating and starting a thread

(and communicating via object)



Break until 8:45

How to do concurrency I - (*all* corners cut!)

At its heart a CPU does this:

```
int PC = start_address;
while (power_on){
    inst = memory[PC++];
    switch (op_code( instr )) {
        case ADD: stack[top] += val(inst); break;
        case JMP: PC = addr(inst); break;
        ...
    }
}
```

How to do concurrency II - (*all* corners cut!)

For concurrency, it does this:

```
int PC = start_address;
int slice = 50;
while (power_on){
    inst = memory[PC++];
    switch (op_code( instr )) {
        case ADD: ...; break;
        case JMP: PC = addr(inst); break;
        ...
    }
    if (slice-- == 0) {
        // save currrent PC, stack pointer, top, etc into list of acticve processe
        // load other process
        slice = 50;
    }
}
```

How to do concurrency II - (*all* corners cut!)

For concurrency, it does this:

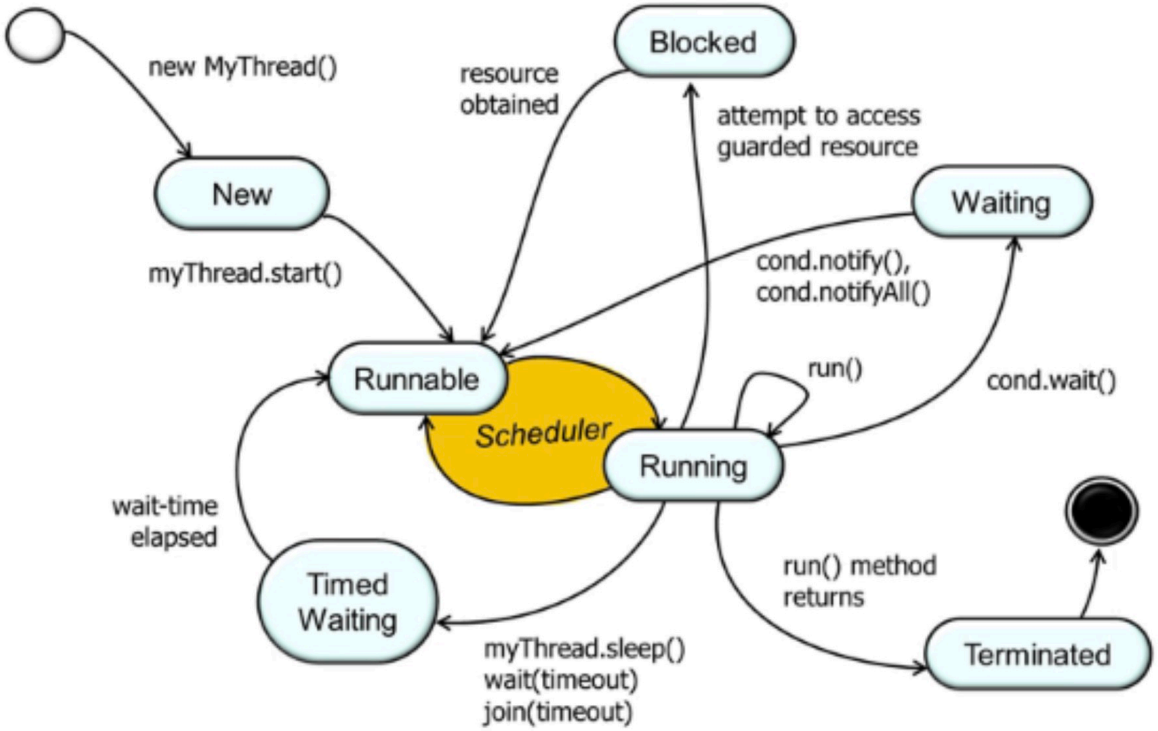
```
int PC = start_address;
int slice = 50;
while (power_on){
    inst = memory[PC++];
    switch (op_code( instr )) {
        case ADD: ...; break;
        case JMP: PC = addr(inst); break;
        ...
    }
    if (slice-- == 0) {
        // save currrent PC, stack pointer, top, etc into list of acticve processe
        // load other process
        slice = 50;
    }
}
```

- A computer with more than N cores, has N such programs
- The cores run in *parallel*, while each core runs several threads *concurrently*
- Each core has its own stack, PC, registers etc.
- All cores share memory

Java virtual machine (JVM)

- Maps each java thread onto a operating system thread
- Java can run as many threads in parallel as there are cores on the computer
- Java compiles all bytecode to native instructions of the computer.
  - Many problems (and advantages) of native threads/processes shows in Java.

Thread states



Taken from https://bitstechnotes.wordpress.com/2017/12/16/java-thread-state-diagram/

Java lambda expressions

Instead of anonymous inner classes:

```
 Thread t = new Thread(
    new Runnable() {
      public void run() {
        while (true)
          lc.increment(); }
}); // TestLongCounter7.java
```

we use lambda expressions:

```
Thread t = new Thread(() -> {
  while (true)
    lc.increment();
}); //TestLongCounter.java
```

Notice: all lambda expressions are made into anonymous classes, and all anonymous classes are given unique hidden names

Locks

Notice: This door does not lock anything.

Locks

Notice: This door does not lock anything.

However, you can use locks to establish conventions for access to resources.

Locks and the synchronized statement

Any Java object (door) can be used for locking

The synchronized statement

```
synchronized (obj) {
    ... body ... }
```

- Blocks until the lock on obj is available
- Takes (acquires) the lock on obj
- Executes the body block
- Releases the lock, also on return or exception (if the body is not run to its end)

By consistently locking on the same object

- one can obtain mutual exclusion, so
- at most one thread can execute the code at a time

A synchronized method locks the “this” reference around body

A synchronized instance method

```
class C {
  public synchronized void method() { ... }
}
```

really uses a synchronized statement:

```
class C {
  public void method() {
    synchronized (this) { ... }
    }
}
```

Q: What is being locked? (The entire class, the method, the instance, the Java system)?

What about synchronized static methods?

A synchronized static method

```
class C {
  public synchronized static void method()
    { ... }
}
```

locks on the class runtime object C.class:

```
class C {
  public static void method() {
    synchronized (C.class) { ... }
    }
}
```

Use explicit lock objects

- So it is clear what object is being locked on
- So only your methods lock on the object

Good:

```
class LongCounter {
    public synchronized void increment() { ... }
    public synchronized long get() { ... }
}
```

Better:

```
class LongCounterBetter {
  private final Object myLock = new Object(); // Explict lock object
  // Only these methods can lock on myLock
  public void increment() {
      synchronized (myLock) { ... }
    }
  public long get() {
        synchronized (myLock) { ... }
  }
}// TestLongCounterBetter.java
```

Multiple threads, locking

Two threads incrementing counter in parallel:

```
final int counts = 10_000_000;
Thread t1 = new Thread(() -> {
    for (int i=0; i < counts; i++)
        lc.increment();
});
Thread t2 = new Thread(() -> {
    for (int i=0; i < counts; i++)
        lc.increment();
}); // TestLongCounterExperiments.java
```

- Q: How many threads are running now?

Starting the threads,

and waiting for their completion

```
t1.start(); t2.start();
```

A thread completes when the lambda returns

- To wait for thread t completing, call t.join()
- May throw InterruptedException

```
try { t1.join(); t2.join(); }
catch (InterruptedException exn) { ... }
System.out.println("Count is " + lc.get());
```

What is lc.get() after threads complete?

- Each thread calls lc.increment() ten million times - So it gets called 20 million times

Removing the locking

Non-thread-safe counter class:

```
class LongCounter2 {
  private long count = 0;
  public void increment() {
    count = count + 1;
  }
  public long get() { return count; }
}
```

- Produces very wrong results, not 20 million:
- Q: Why?

```
Count is 10041965
Count is 19861602
Count is 18939813
```

The operation

`count = count + 1 is not atomic`

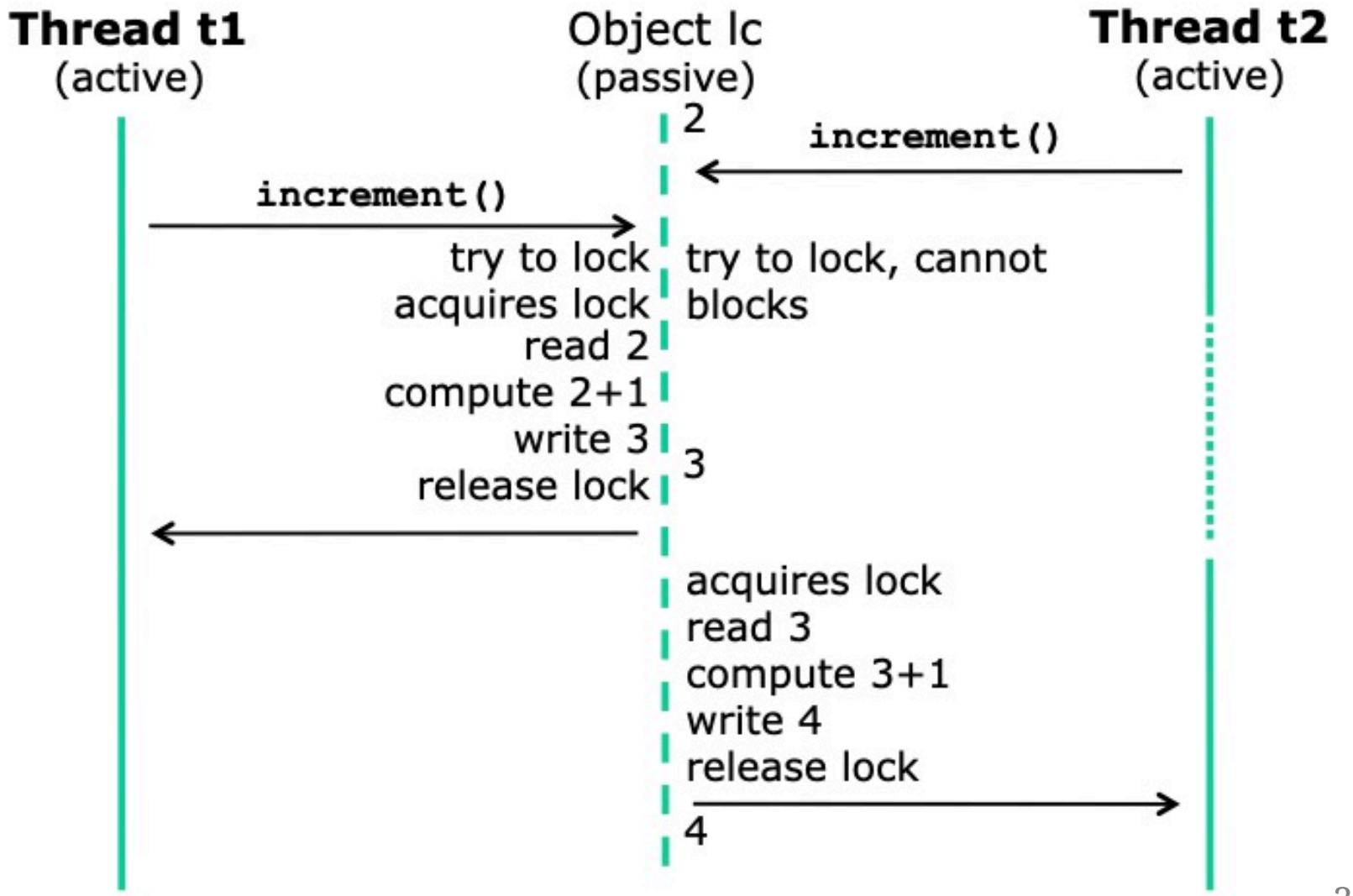
- What count = count + 1 means:
  - read count
  - add 1
  - write result to count
- Hence not atomic

So risk that two increment() calls will increase count by only 1

- NB: Same for `count += 1` and `count++`

No locking: lost update

Mutual exclusion (using locking)



Break (maybe)

Using threads for performance

Example: Count primes 2 3 5 7 11 ...

Count primes in 0...9999999:

```
static long countSequential(int range) {
  long count = 0;
  final int from = 0, to = range;
  for (int i=from; i < to; i++)
    if (isPrime(i))
      count++;
  return count;
} // TestCountPrimes.java
```

Result is `664579`

- Takes 6.4 sec to compute on 1 CPU core
- Why not use all my computer's 4 (x 2) cores?
  - Eg. use two threads t1 and t2 and divide the work: t1: 0...4999999 and t2: 5000000...9999999

Using two threads to count primes

```
final LongCounter lc = new LongCounter();
final int from1 = 0, to1 = perThread;
Thread t1 = new Thread(() -> {
  for (int i=from1; i < to1; i++)
    if (isPrime(i))
        lc.increment();
});
final int from2 = perThread, to2 = perThread * 2;
Thread t2 = new Thread(() -> {
   for (int i=from2; i < to2; i++)
    if (isPrime(i))
        lc.increment();
}); // TestCountPrimes.java
```

Same code twice, bad practice

- Takes 4.2 sec real time, so already faster
- Q: Why not just use a long count variable?
- Q: What if we want to use 10 threads?

Using N threads to count primes

```
final LongCounter lc = new LongCounter();
Thread[] threads = new Thread[threadCount];
for (int t=0; t < threadCount; t++) {
  final int from = perThread * t,
    // Last thread has to == range
    to = (t+1 == threadCount) ? range : perThread * (t+1);
  threads[t] = new Thread(() -> { // Thread processes segment [from,to)
    for (int i=from; i < to; i++)
      if (isPrime(i))
        lc.increment();
  });
}
for (int t=0; t < threadCount; t++) threads[t].start();
```

Takes 1.8 sec real time with threadCount 10

- Approx 3.3 times faster than sequential solution
- Q: Why not 4 times, or 10 times faster?

Reflections: threads for performance

- This code can be made better in many ways
  - Eg better distribution of work on the 10 threads
  - Eg less use of the synchronized LongCounter
- Use Java 8 parallel streams instead *later in course*
- Proper performance measurements, *later in course*
- Very bad idea to use many (> 500) threads
  - Each thread takes much memory for the stack
  - Each thread slows down the garbage collector
- Use tasks and Java “executors”, *later in course*
- More advice on scalability, *later in course*
- How to avoid locking *later in course*

A side remark - on `final`

Why do we use `final`?

Consider this piece of code (pseudo java):

```
public class LambdaFail {
  public static void main(String[] args) {
    IntSupplier myLambda = incrementer(10);
        System.out.println("First value: " + myLambda );
        System.out.println("Second value: " + myLambda );
    }

    private static IntSupplier incrementer(int start){
        return ()->start++;
    }
} // LambdaFail.java
```

A side remark - on `final`

Why do we use `final`?

Consider this piece of code (pseudo java):

```
public class LambdaFail {
  public static void main(String[] args) {
    IntSupplier myLambda = incrementer(10);
        System.out.println("First value: " + myLambda );
        System.out.println("Second value: " + myLambda );
    }

    private static IntSupplier incrementer(int start){
        return ()->start++;
    }
} // LambdaFail.java
```

- If java allowed this, `mylambda` would in effect be able to refer to a parameter of a returned method (the `start` parameter).
- This is a problem which has to do with lambda, not threads.
- But we use lambda to start threads, hence it affects us in practice.