Practical Concurrent and Parallel Programming 1

Peter Sestoft
IT University of Copenhagen

Friday 2016-09-02

Plan for today

- Why this course?
- Course contents, learning goals
- Practical information
- Mandatory exercises, examination
- Java threads
- Java locking, the synchronized keyword
 - Use synchronized on blocks, not on methods
- Visibility of memory writes
- Threads for performance

The teachers

- Course responsible: Peter Sestoft
 - MSc 1988 and PhD 1991, Copenhagen University

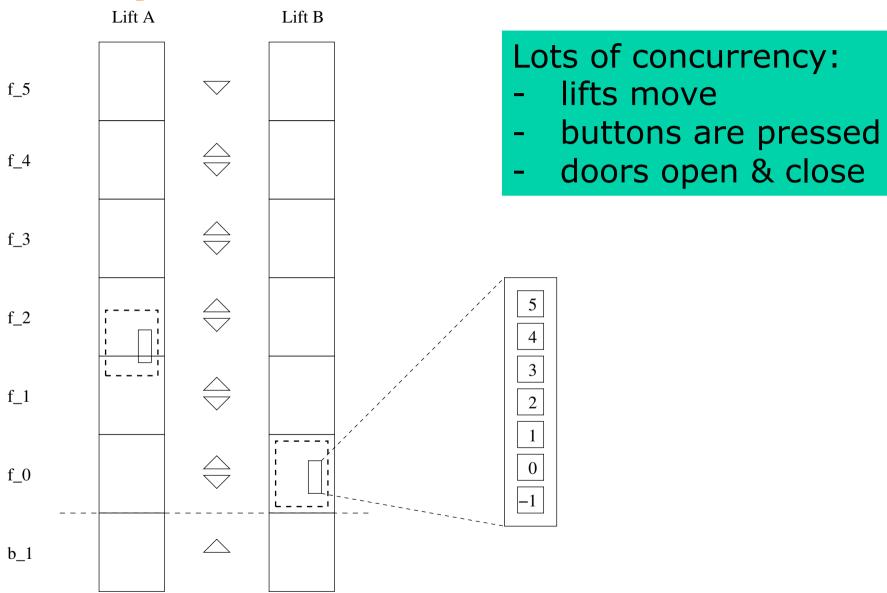


- Co-teacher: Claus Brabrand
- Exercises
 - Alexander Bock, ITU PhD student, ITU MSc graduate
 - Gediminas Kucas, ITU MSc student
 - Sigurt Bladt Dinesen, ITU MSc student
 - (Amund Ranheim Lome, ITU MSc graduate)

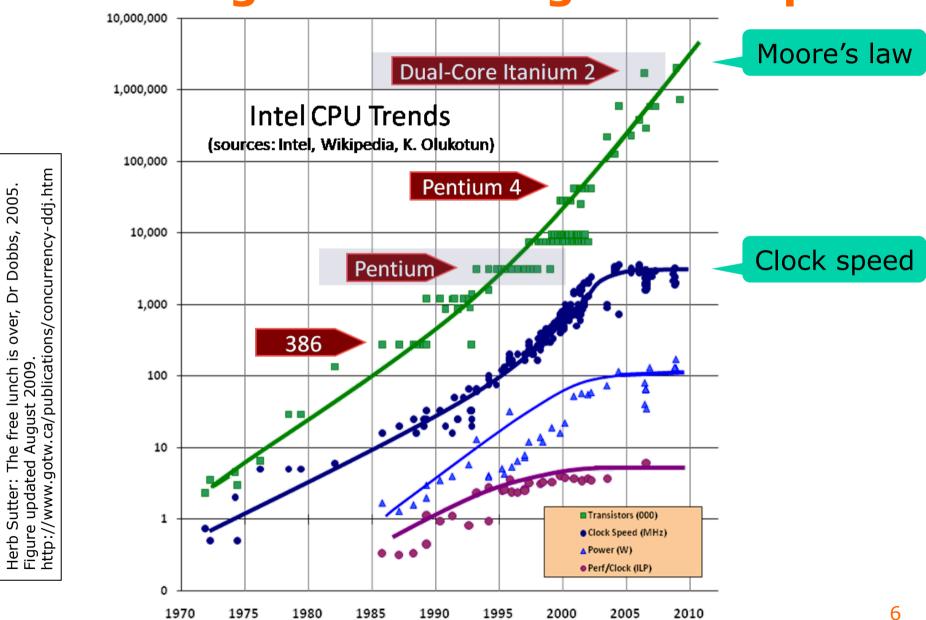
Why this course?

- Parallel programming is necessary
 - For responsiveness in user interfaces etc.
 - 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: The free lunch is over
- 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

Example: 2 lifts, 7 floors, 26 buttons



The free lunch is over: No more growth in single-core speed



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
- Transactional memory, Multiverse
- Lock-free data structures, Java mem. model
- Message passing, Akka

Learning objectives

After the course, the successful student can:

- ANALYSE the correctness of concurrent Java software, and RELATE it to the Java memory model
- ANALYSE the performance of concurrent Java software
- APPLY Java threads and related language features (locks, final and volatile fields) and libraries (concurrent collections) to CONSTRUCT correct and well-performing concurrent Java software
- USE software tools for accelerated testing and analysis of concurrency problems in Java software
- CONTRAST different communication mechanisms (shared mutable memory, transactional memory, message passing)

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 review the basics of
 - Java threads
 - Java synchronized methods and statements
 - Java's final keyword
 - Java inner classes

Standard Friday plan

- Fridays until 9 December (except 21 Oct)
- Lectures 0800-1000 in auditorium 1
- Exercise startup
 - 1000-1200 in auditorium 1 and 2A54 and 4A54
- Exercise hand-in: 6.5 days after lecture
 - That is, the following Thursday at 23:55

Course information online

- Course LearnIT page, restricted access: https://learnit.itu.dk/course/view.php?id=3016179
 - Exercises and hand-ins, deadlines, feedback
 - Mandatory exercises and hand-ins, deadlines, feedback
 - Discussion forum
 - Non-public reading materials
- Course homepage, public access: http://www.itu.dk/people/sestoft/itu/PCPP/E2016/
 - Overview of lectures and exercises
 - Lecture slides and exercise sheets
 - Example code
 - List of all mandatory reading materials

Exercises

- There are 13 sets of weekly exercises
- Hand in the solutions through LearnIT
- You can work in teams of 1 or 2 students
- The teaching assistants provide joint feedback
- Six of the 13 weekly exercise sets are mandatory
- At least five of those must be 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
 - What is important is that you learn

The exam

- A 30 hour take-home written exam/project
 - Start at 0900 Tuesday 10 January 2017
 - End at 1500 Wednesday 12 January at 1500
 - 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 (January 2015 and 2016) exams are on the public homepage

Stuff you need

- 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
- Various optional materials, see homepage:
 - Bloch: Effective Java, 2008, highly recommended
 - Sestoft: Java Precisely, 3rd edition 2016
 - more ...

What about other languages?

- .NET and C# are very similar to Java
 - We will point out differences on the way
- Clojure, Scala, F#, ... build on JVM or .NET
 - So thread concepts are very similar too
- C and C++ have some differences (ignore)
- Haskell has transactional memory
 - We will see this in Java too (Multiverse)
- Erlang, Scala, F# have message passing
 - We will see this in Java too (Akka)
- Dataflow, CSP, CCS, Pi-calculus, Join, Cω, ...
 - Zillions of other concurrency mechanisms

Other concurrency models

- Java threads interact via shared mutable fields
 - Shared: Visible to multiple threads
 - Mutable: The fields can be updated, assigned to
- This is a source of many problems
- Alternatives exist:
- No sharing: interact via message passing
 - Erlang, Scala, MPI, F#, Go ... and Java Akka library
- No mutability: use functional programming
 - Haskell, F#, ML, Google MapReduce, ...
- Allow shared mutable mem., but avoid locks
 - Transactional memory, optimistic concurrency
 - In Haskell, Clojure, ... and Java Multiverse library

Other parallel hardware

- We focus on multicore (standard) hardware
 - Typically 2-32 general cores on a CPU chip
 - (Instruction-level parallelism, invisible to software)
- Other types of parallel hardware exist
- Vector instructions (SIMD, SSE, AVX) on core
 - Typically 2-8 floating-point operations/CPU cycle
 - Soon available through .NET JIT and hence C#
- General purpose graphics processors GPGPU
 - Such as Nvidia CUDA, up to 2500 cores on a chip
 - We're using those in a research project
- Clusters, cloud: servers connected by network

Threads and concurrency in Java

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

TestLongCounter.java

A thread-safe class for counting

A thread-safe long counter:

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

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

A thread that increments the counter

- A Thread t is created from a Runnable
- The thread's behavior is in the run method

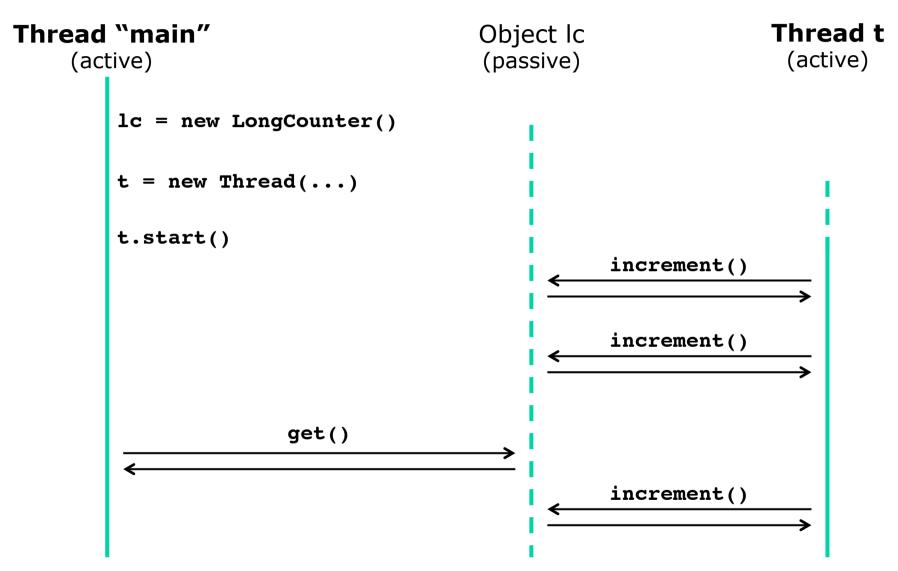
```
NB!
final LongCounter lc = new LongCounter();
Thread t =
                                          An anonymous inner
   new Thread (
                                             class, and an
     new Runnable()
                                             instance of it
        public void run()
           while (true)
                                           When started, the
             lc.increment();
                                           thread will do this:
                                           increment forever
                                                             TestLongCounter7.jav
```

- This only creates the thread, does not start it
- Q: What does **final** mean?

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() { ... });
  t.start();
  System.out.println("Press Enter ... ");
  while (true) {
    System.in.read();
    System.out.println(lc.get());
  }
}
```

Creating and starting a thread (and communicating via object)



Java 8 lambda expressions

Instead of old anonymous inner classes:

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

• ... we use neat Java 8 lambda expressions:

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

Locks and the synchronized statement

- Any Java object 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
- 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 simply 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 void method() {
    synchronized (C.class) { ... }
  }
}
```

Use synchronized statements, not synchronized methods

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

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

Only these methods can lock on myLock

```
class LongCounterBetter {
  private final Object myLock = new Object();
  public void increment() {
    synchronized (myLock) { ... }
  }
  public long get() {
    synchronized (myLock) { ... }
  }
}
Clear what
  is locked on
Better
```

TestLongCounterExperiments.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();
});</pre>
```

Q: How many threads are running now?

Starting the threads, and waiting for their completion

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

- A thread completes when run 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:

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

• Q: Why?

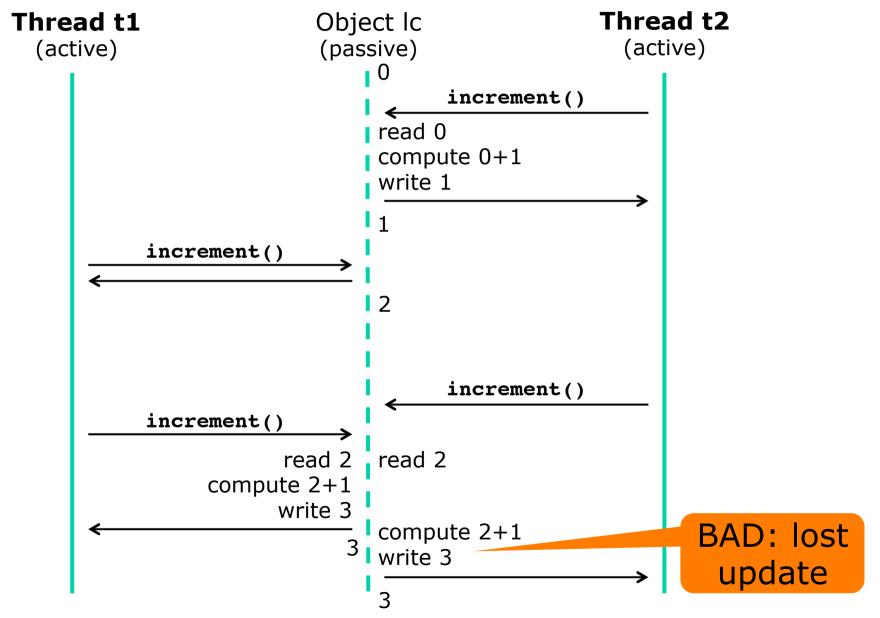
The operation count = count + 1 is not atomic

- 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++

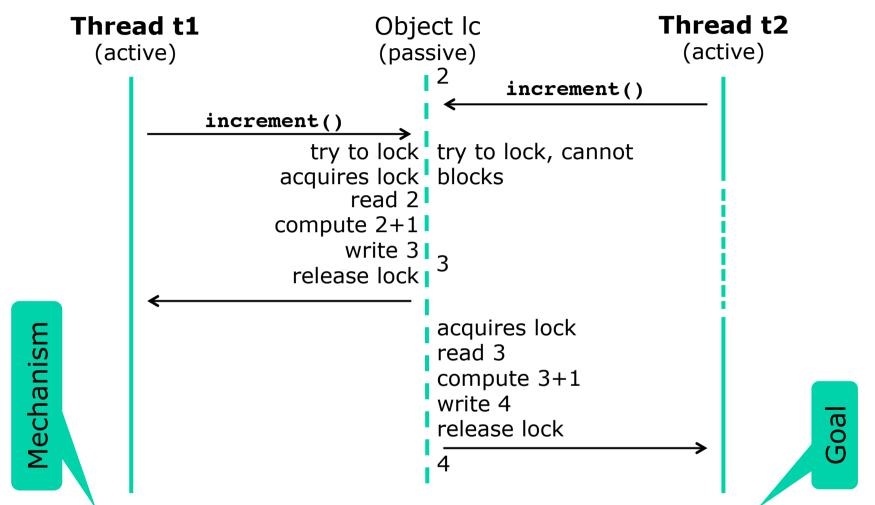
Without locking

No locking: lost update



How does locking help?

With locking



- Locking can achieve mutual exclusion
 - When used on all state accesses
 - Unfortunately, quite easy to get it wrong

Why synchronize just to read data?

```
class LongCounter {
  private long count = 0;
  public synchronized void increment() {
    count = count + 1;
  }
  public synchronized long get() {
    return count;
  } }
Why needed?
```

- The synchronized keyword has two effects:
 - Mutual exclusion: only one thread can hold a lock (execute a synchronized method or block) at a time
 - Visibility of memory writes: All writes by thread A before releasing a lock (exit synchr) are visible to thread B after acquiring the lock (enter synchr)

Visibility is really important

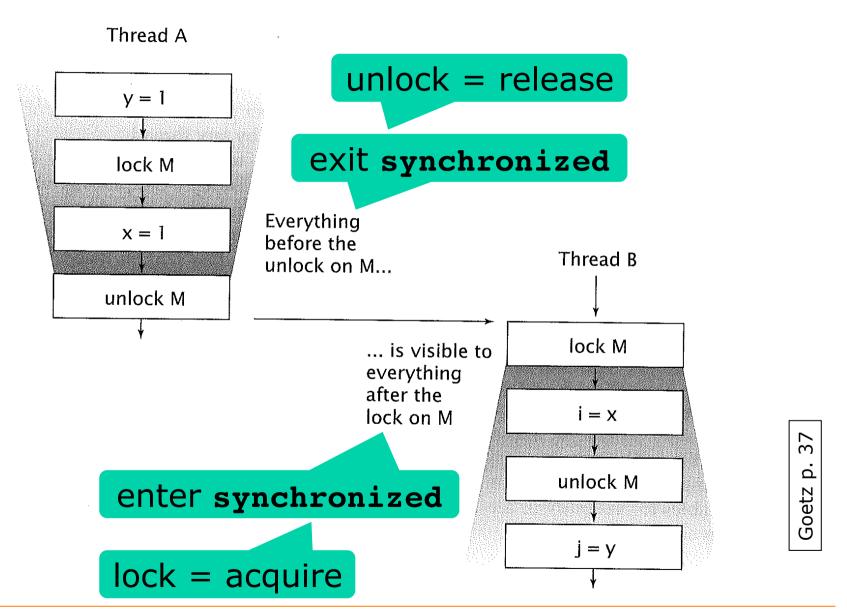
```
class MutableInteger {
    private int value = 0;
    public void set(int value) { this.value = value; }
    public int get() { return value; }
}
```

- Looks OK, no need for synchronization?
- But thread t may loop forever in this scenario:

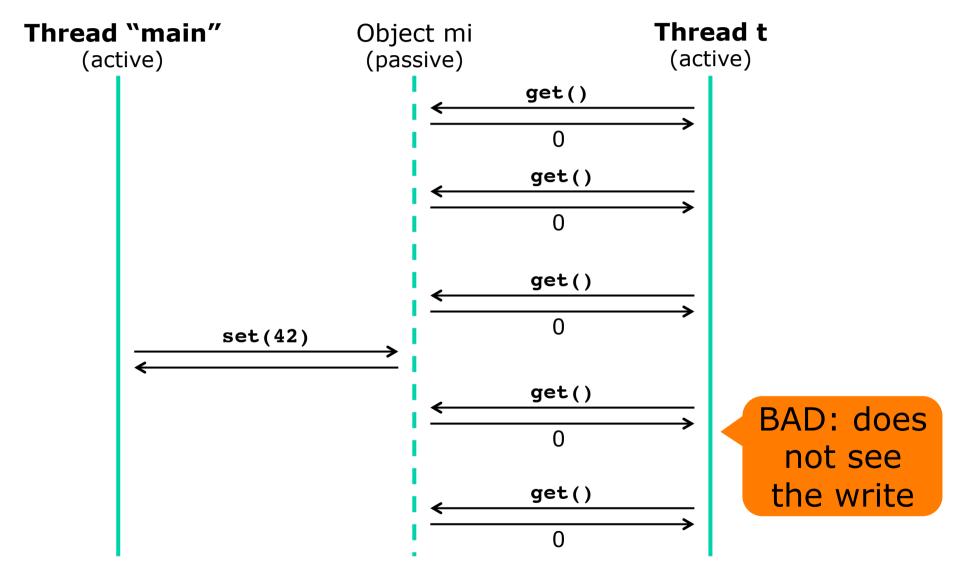
```
final MutableInteger mi = new MutableInteger();
Thread t = new Thread(() -> {
    while (mi.get() == 0) { }
});
t.start();
mi.set(42);
This write by thread "main" may be forever invisible to thread t
```

- Two possible fixes:
 - Add synchronized to methods get and set, OR
 - Add volatile to field value

Visibility by synchronization



Communication through mutable shared state fails if no visibility



The volatile field modifier

 The volatile field modifier can be used to ensure visibility (but not mutual exclusion)

```
class MutableInteger {
   private volatile int value = 0;
   public void set(int value) { this.value = value; }
   public int get() { return value; }
}
```

- All writes by thread A before writing a
 volatile field are visible to thread B when,
 and after, reading the volatile field
- Note: A single volatile write+read makes writes to all other fields visible also!
 - A bit mysterious, but a consequence of the implementation
 - This is Java semantics; C#, C, C++ volatile are different

Goetz p. 38, 39

Goetz advice on volatile

Use volatile variables only when they simplify your synchronization policy; avoid it when verifying correctness would require subtle reasoning about visibility.

Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility.

- Rule 1: Use locks (synchronized)
- Rule 2: If circumstances are right, and you are an expert, maybe use volatile instead
- Rule 3: There are few experts

That was Java. What about C# and .NET?

- C# Language Spec. §17.3.4 Volatile Fields
- CLI Ecma-335 standard section §I.12.6.7:
 - "A volatile write has release semantics ... the write is guaranteed to happen after any memory references prior to the write instruction in the CIL instruction sequence"
 - "volatile read has acquire semantics ... the read is guaranteed to occur prior to any references to memory that occur after the read instruction in the CIL instruction sequence"
- C#'s volatile is weaker than Java's
 - And very unclearly described
 - Maybe use C# lock or MemoryBarrier() instead

Ways to ensure visibility

- Unlocking followed by locking the same lock
- Writing a volatile field and then reading it
- Calling one method on a concurrent collection and another method on same collection
 - java.util.concurrent.*
- Calling one method on an atomic variable and then another method on same variable
 - java.util.concurrent.atomic.*
- Finishing a constructor that initializes final or volatile fields
- Calling t.start() before anything in thread t
- Anything in thread t before t.join() returns

(Java Language Specification 8 §17.4, and the Javadoc for concurrent collection classes etc, give the full and rather complicated details)

TestCountPrimes.java

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;
}</pre>
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++)</pre>
    if (isPrime(i))
      lc.increment();
});
final int from2 = perThread, to2 = perThread * 2;
Thread t2 = new Thread(() -> {
  for (int i=from2; i<to2; i++)</pre>
                                        Same code twice,
    if (isPrime(i))
                                          bad practice
      lc.increment();
});
```

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

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, week 3
- Proper performance measurements, week 4
- 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", week 5
- More advice on scalability, week 7
- How to avoid locking, week 10 and 11

Why "concurrent" and "parallel"?

- Informally both mean "at the same time"
- But some people distinguish
 - Concurrent: related to correctness
 - Parallel: related to performance
- Soccer (fodbold) analogy, by P. Panangaden
 - The referee (dommer) is concerned with concurrency: the soccer rules must be followed
 - The coach (*træner*) is concerned with parallelism: the best possible use of the team's 11 players
- This course is concerned with correctness as well as performance: concurrent and parallel

Processes, threads, and tasks

- An operating system process running Java is
 - a Java Virtual Machine that executes code
 - an object heap, managed by a garbage collector
 - one or more running Java threads
- A Java thread
 - has its own method call stack, takes much memory
 - shares the object heap with other threads
- A **task** (or future) (or actor)
 - does not have a call stack, so takes little memory
 - is run by an executor, using a thread pool, week 5

This week

- Reading
 - Goetz chapters 1, 2 and 3
 - Sutter paper
 - Bloch item 66
- Exercises week 1, on homepage and LearnIT
 - Make sure you are familiar with Java threads and locks and inner classes
 - Make sure that you can compile, run and explain programs that use these features
- Read before next week's lecture
 - Goetz chapters 4 and 5
 - Bloch item 15