Practical Concurrent and Parallel Programming 1

Peter Sestoft
IT University of Copenhagen

Friday 2014-08-29**

Plan for today

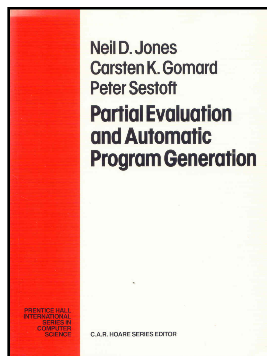
- Why this course?
- Course contents, learning goals
- Practical information
- Mandatory exercises, examination

- Java threads
- Java inner classes
- Java locking, the **`synchronized`** keyword

- Quizzes

The teachers

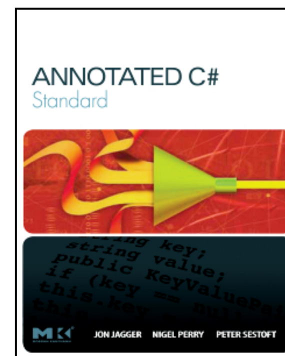
- Course responsible: Peter Sestoft
  - MSc 1988 and PhD 1991, Copenhagen University



1993

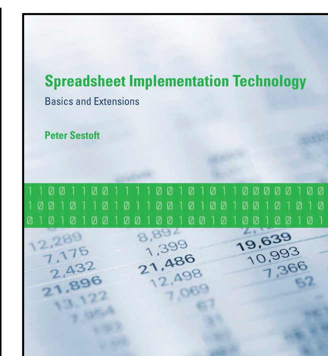
2002 & 2005

2004 & 2012



2007

2012



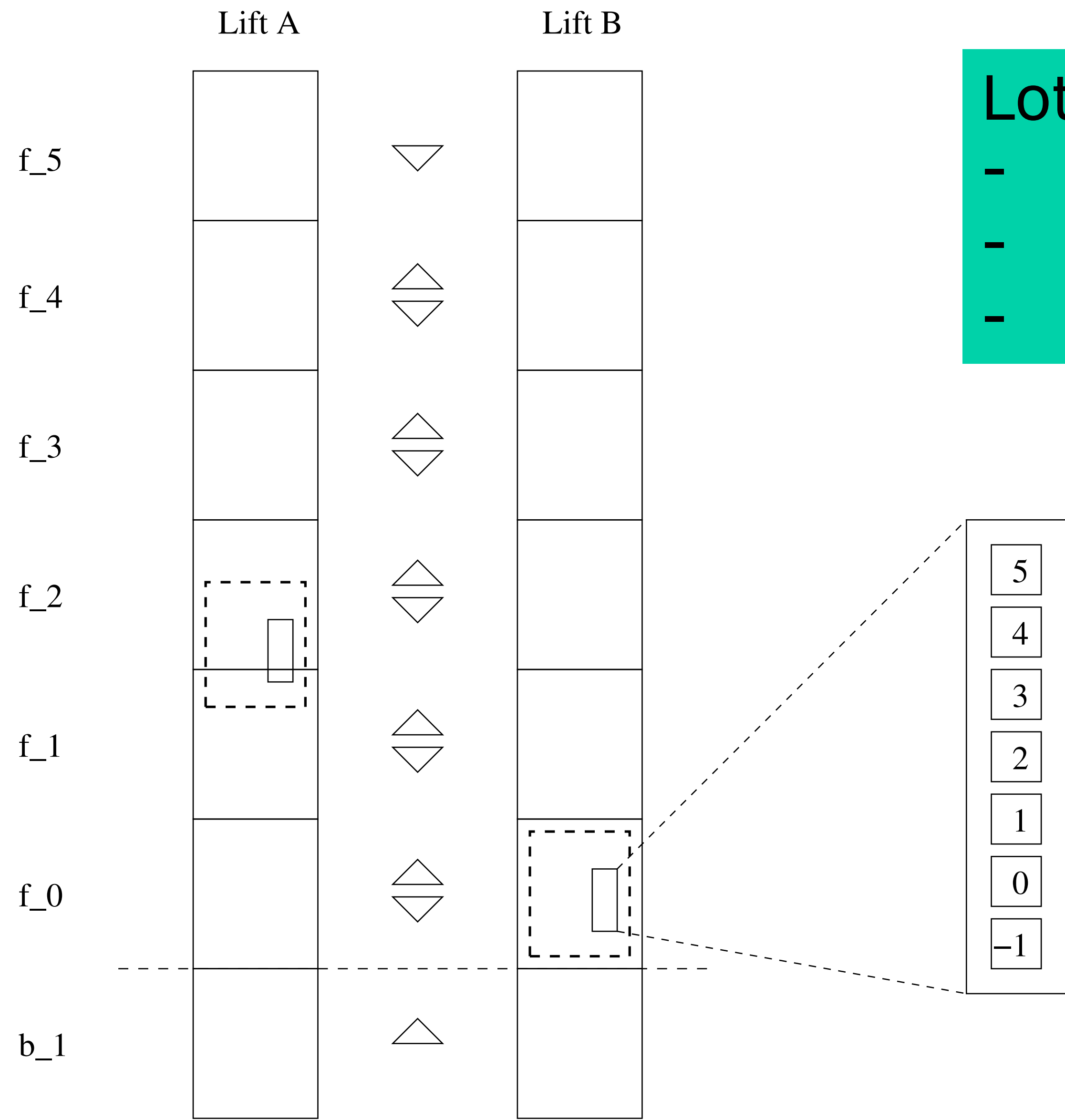
2014

- Co-teacher: Claus Brabrand
- Exercises
  - Iago Abal Rivas, ITU PhD student
  - Florian Biermann, research assistant, ex-ITU MSc
  - Håkan Lane, PhD, external teaching assistant

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

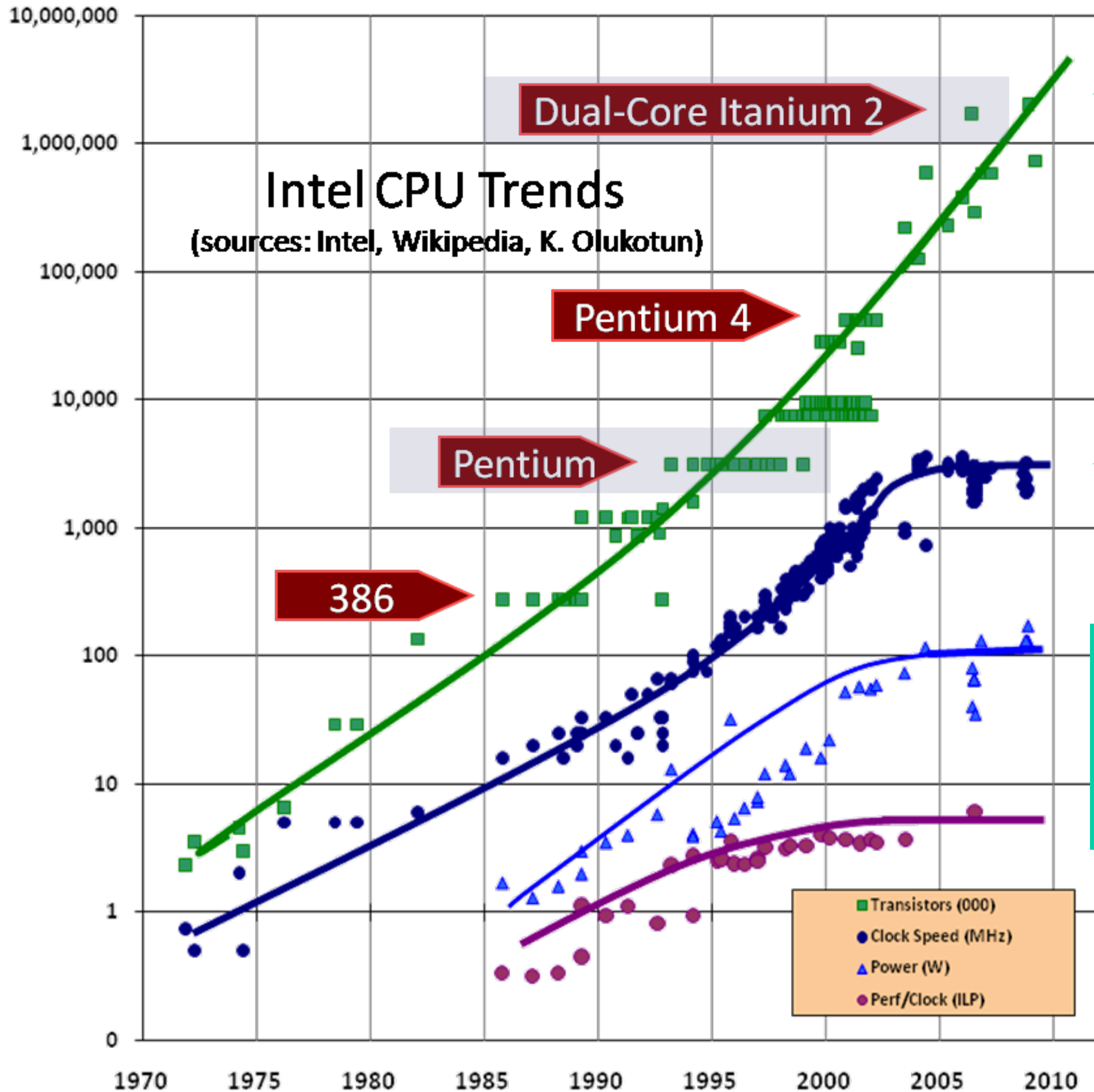
Example: 2 lifts, 7 floors, 26 buttons



Lots of concurrency:

- lifts move
- buttons are pressed
- doors open & close

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



Moore’s law

Clock speed

At 3 GHz light travels 10 cm/cycle

Herb Sutter: The free lunch is over, Dr Dobbs, 2005.
Figure updated August 2009.
http://www.gotw.ca/publications/concurrency-ddj.htm

Course contents

- Threads, locks, mutual exclusion, scalability
- Performance measurements
- Tasks, the Java executor framework
- Safety, liveness, deadlocks
- Testing concurrent programs, ThreadSafe
- 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 5 December (except 17 Oct)
- Lectures 0800-1000
- Exercise startup
  - either 1000-1200 (Iago, Florian)
  - or 1200-1400 (Håkan)
- 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=3000701
  - 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/SPPP/E2014/
  - 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 will provide 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 [CHANGED 12 SEP]

- A 37 hour take-home written exam/project
  - Start at 0900 on Wednesday 7 January 2015
  - End at 2200 on Thursday 8 January
  - Electronic submission
- 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

Stuff you need

- Goetz et al: *Java Concurrency in Practice*
  - From 2006, still the best on Java concurrency
  - Almost everything is relevant for C#/.NET too
- Free lecture notes and papers, see homepage
- A few other book chapters, see LearnIT

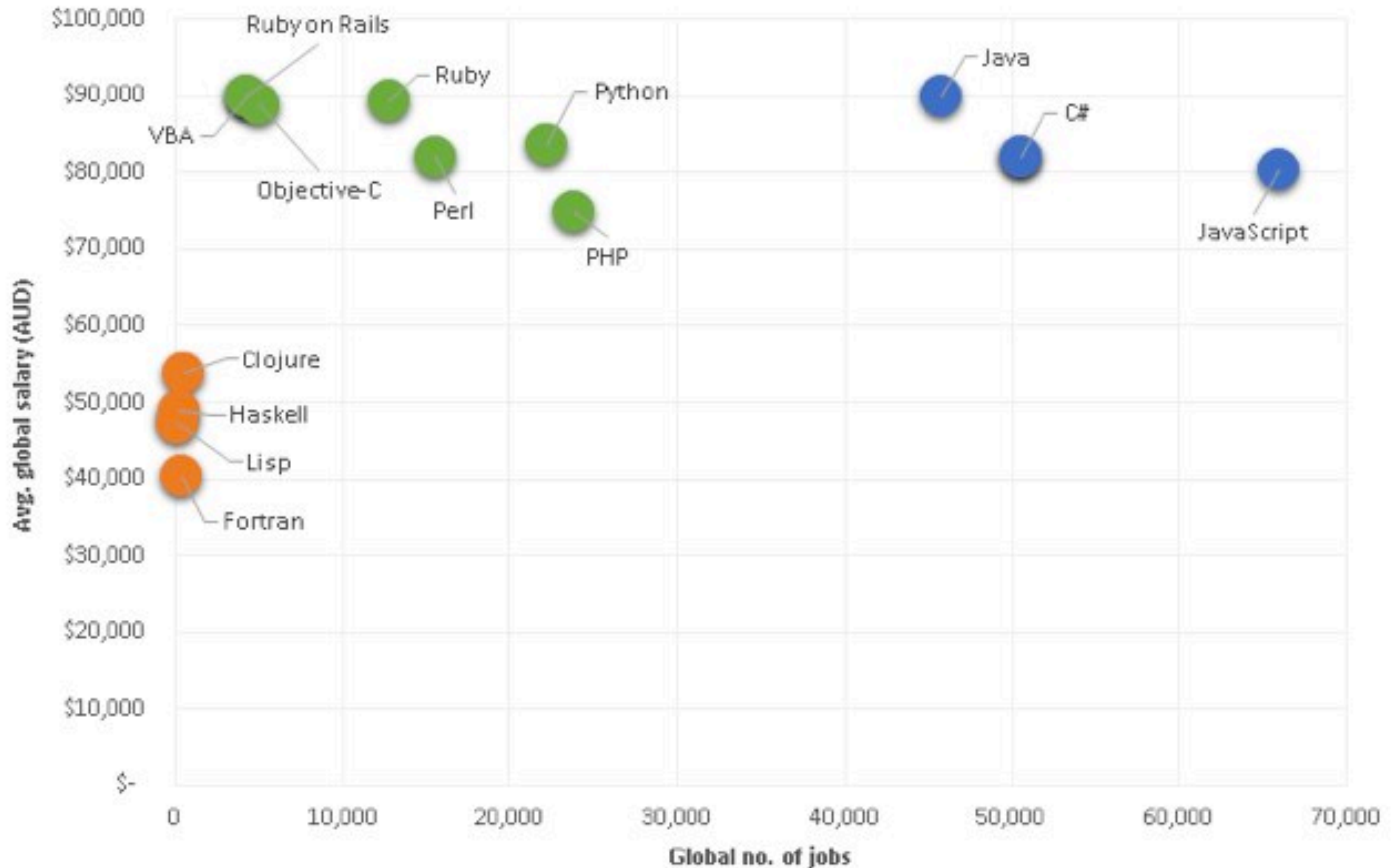
- Java 7 or 8 SDK installed on your computer

- Various optional materials, see homepage:
  - Bloch: *Effective Java*, 2008, **highly recommended**
  - Sestoft: *Java Precisely*, 2005
  - 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

Salary and jobs by language



https://msgooroo.com/GoorooTHINK/Article/16191/Which-language-wins-in-terms-of-salarydemand-July-2014

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**

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;
  }
}
```

TestLongCounter.java

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

An anonymous inner class, and an instance of it

When started, the thread will do this: increment forever

- 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());
  }
}
```

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

Creating and starting a thread

Thread "main"
(active)

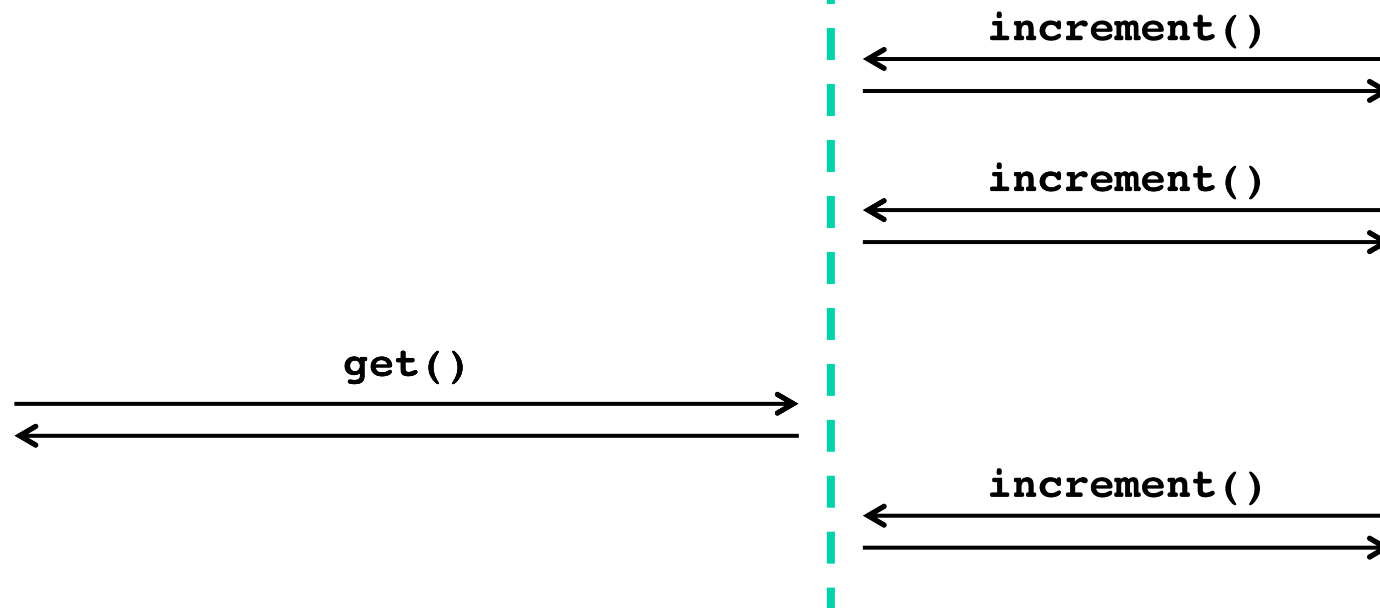
Object lc
(passive)

Thread t
(active)

```
lc = new LongCounter()

t = new Thread(...)

t.start()
```



Java (1-7) anonymous inner classes

- A statement must be in a method, eg. **`run`**

```
public interface Runnable {
  public void run();
}
```

- An anonymous inner class is a quick way to
  - create a class that implements Runnable *and*
  - create an instance of it

Anonymous inner class and instance

```
Runnable r =
  new Runnable() {
    public void run() {
      ... some code we want to execute ...
    }
  };
```

Locks and the synchronized keyword

- 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 is just one with a synchronized 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 *static* synchronized methods?

- A synchronized static method in class C

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

locks on the reflected Class object for C:

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

- Not important to understand **`C.class`** because
  - Dangerous to share static fields between threads
    - Except possibly in factory methods

Multiple threads, locking

- Two threads incrementing counter in parallel:

```
final int counts = 10_000_000;
Thread t1 = new Thread(new Runnable() { public void run() {
  for (int i=0; i<counts; i++)
    lc.increment();
}});
Thread t2 = new Thread(new Runnable() { public void run() {
  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 **`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

`count = count + 1` means:
read **count**, add 1, write result to **count**

- Possible scenario when **count** is 42, and two threads call **lc.increment()** at the same time

Thread 1	Thread 2
Read 42 from count	
	Read 42 from count
Add 1, giving 43	
	Add 1, giving 43
Write 43 to count	
	Write 43 to count

- Two increments but **count** only increased by 1
- So-called *lost update*

Why does locking help?

Thread 1	Thread 2
Try to lock, get lock	
Read 42 from count	
	Try to lock, cannot, must block
Add 1, giving 43	(blocked)
Write 43 to count	(blocked)
Release lock	(blocked)
	Get lock
	Read 43 from count
	Add 1, giving 44
	Write 44 to count
	Release lock

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

Reads must be synchronized also

- A very small bank with two accounts:

```
class Bank {
  private long account1 = 3000, account2 = 2000;
  public synchronized void transfer(long amount) {
    account1 -= amount;
    account2 += amount;
  }
  public synchronized long getSum() {
    return account1 + account2;
  }
}
```

Transfer, should preserve sum

Count sum of bank's money

TestAccountTransfer.java

- Common mistake to believe that only writes, method **`transfer`**, must be synchronized
- But we need **`synchronized`** on **both** methods!

Transferring money, and counting it, at the same time

- Transfer money, and concurrently count it:

```
final Bank bank = new Bank();
final int transfers = 10_000_000;
final Thread clerk = new Thread(new Runnable() {
  public void run() {
    for (int i=0; i<transfers; i++)
      bank.transfer(rnd.nextInt(10000));
  }});
clerk.start();
for (int i=0; i<100; i++)
  System.out.println(bank.getSum());
```

Make many transfers

Print sum

- Q: Why must **both** Bank methods be synchr.?
- Even a single field read must be synchronized
  - But for another reason, see next week's lecture

Transferring money concurrently

- With a single clerk, final sum is 5000 always
- With two clerks, the final sum may be wrong
  - When **`transfer`** is not synchronized

```
final Thread clerk1 = new Thread(new Runnable() {
  public void run() {
    for (int i=0; i<transfers; i++)
      bank.transfer(rnd.nextInt(10000));
    }});
final Thread clerk2 = ... exact same code ...
clerk1.start(); clerk2.start();
```

- Q: What scenario may give wrong final sum?
- Simplified take-home message:
  - **All** reads and writes must be synchronized

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

This week

- Reading
  - Goetz chapters 1 and 2
  - 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 2 and 3
  - Bloch item 15