Zaawansowane programowanie obiektowe

Lecture 5 (concurrent programming)

Szymon Grabowski
sgrabow@kis.p.lodz.pl
http://szgrabowski.kis.p.lodz.pl/zpo18/

Łódź, 2018

Processes and threads

Processes are tasks that a given OS runs (in parallel).
A process has a self-contained execution environment:
a complete, private set of basic run-time resources;
in particular, each process has its own memory space.

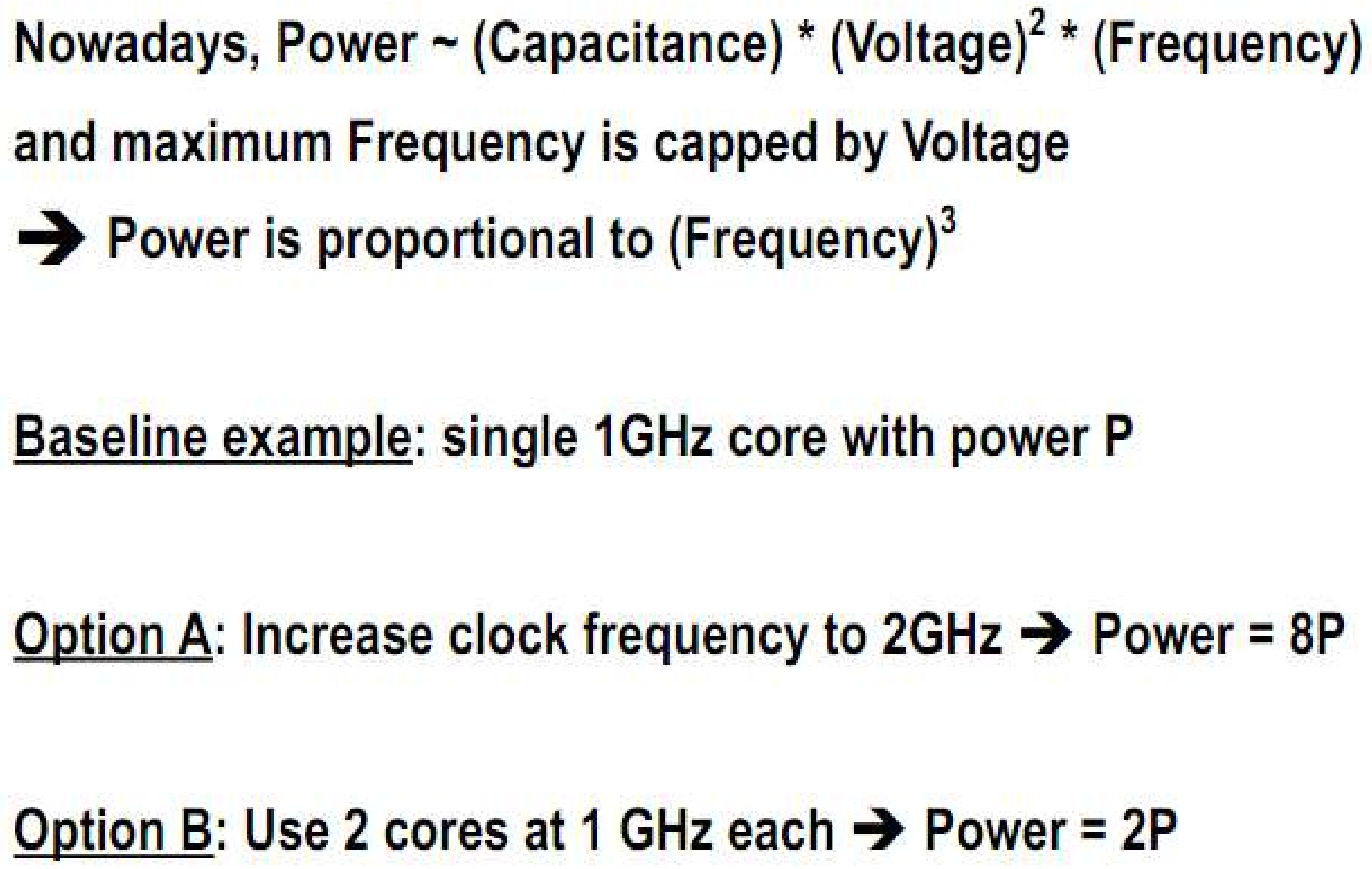
Threads are “sub-processes” that run concurrently
within a single process.

Java provides language-level and library support for threads —
independent sequences of execution within the same program
that share the same code and data address space. Each thread
has its own runtime stack to make method calls
and store local variables.

Most stuff in this lecture based on
http://www.cs.stir.ac.uk/courses/31V4/lectures/
But see also
http://www.cs.usfca.edu/~parrt/doc/java/Threads-notes.pdf

Parallelism saves power

Simplified analysis:



Nowadays, Power ~ (Capacitance) * $(\text{Voltage})^2$ * (Frequency)

and maximum Frequency is capped by Voltage

➔ Power is proportional to $(\text{Frequency})^3$

Baseline example: single 1GHz core with power P

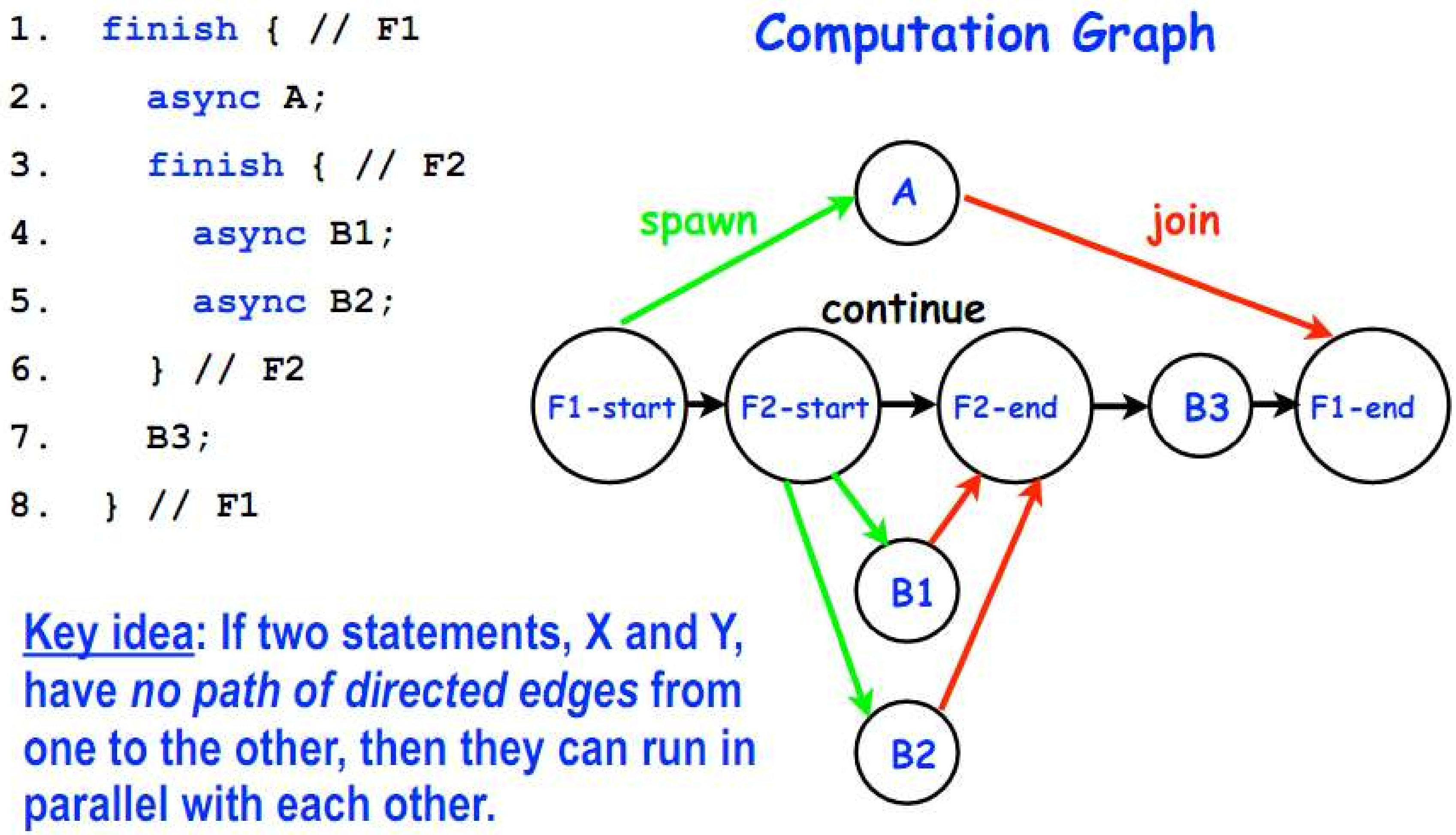
Option A: Increase clock frequency to 2GHz ➔ Power = 8P

Option B: Use 2 cores at 1 GHz each ➔ Power = 2P

http://www.cs.rice.edu/~vsarkar/PDF/comp322-s15-lec1-slides-v1.pdf

Computation graph

[https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s15-lec2-slides-v1.pdf]



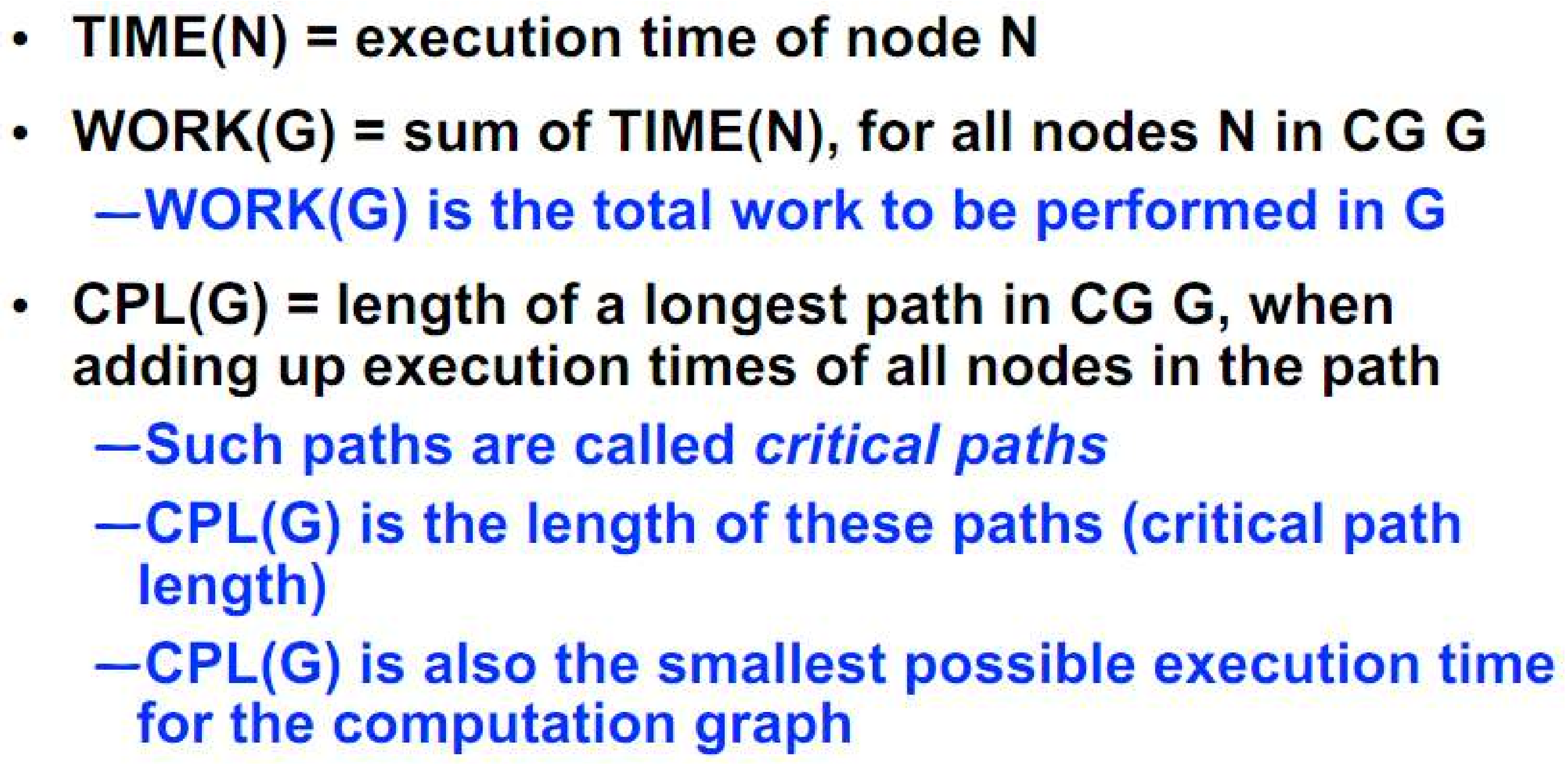
```
finish { // F1
  async A;
  finish { // F2
    async B1;
    async B2;
  } // F2
  B3;
} // F1
```

Computation Graph

Key idea: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Complexity measures for computation graphs

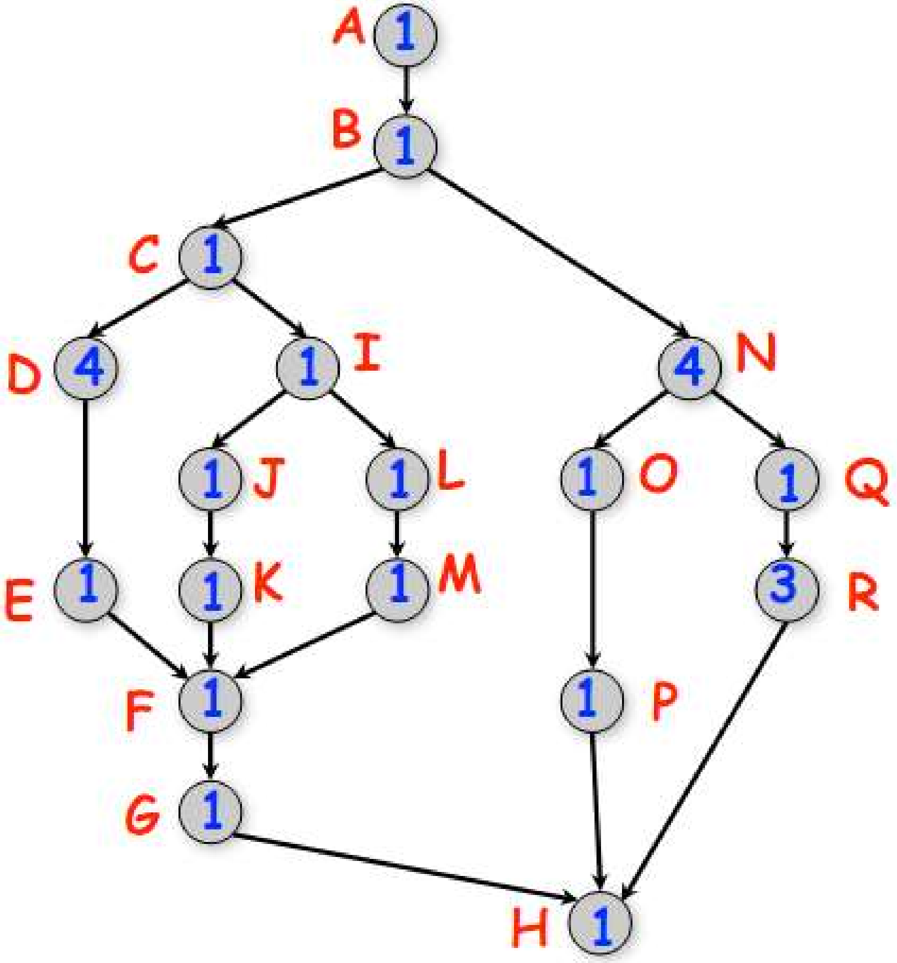
[https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s15-lec2-slides-v1.pdf]



- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called *critical paths*
  - CPL(G) is the length of these paths (critical path length)
  - CPL(G) is also the smallest possible execution time for the computation graph

Scheduling, fixed # of processors (example)

[https://wiki.rice.edu/confluence/display/PARPROG/COMP322 , lecture 3]



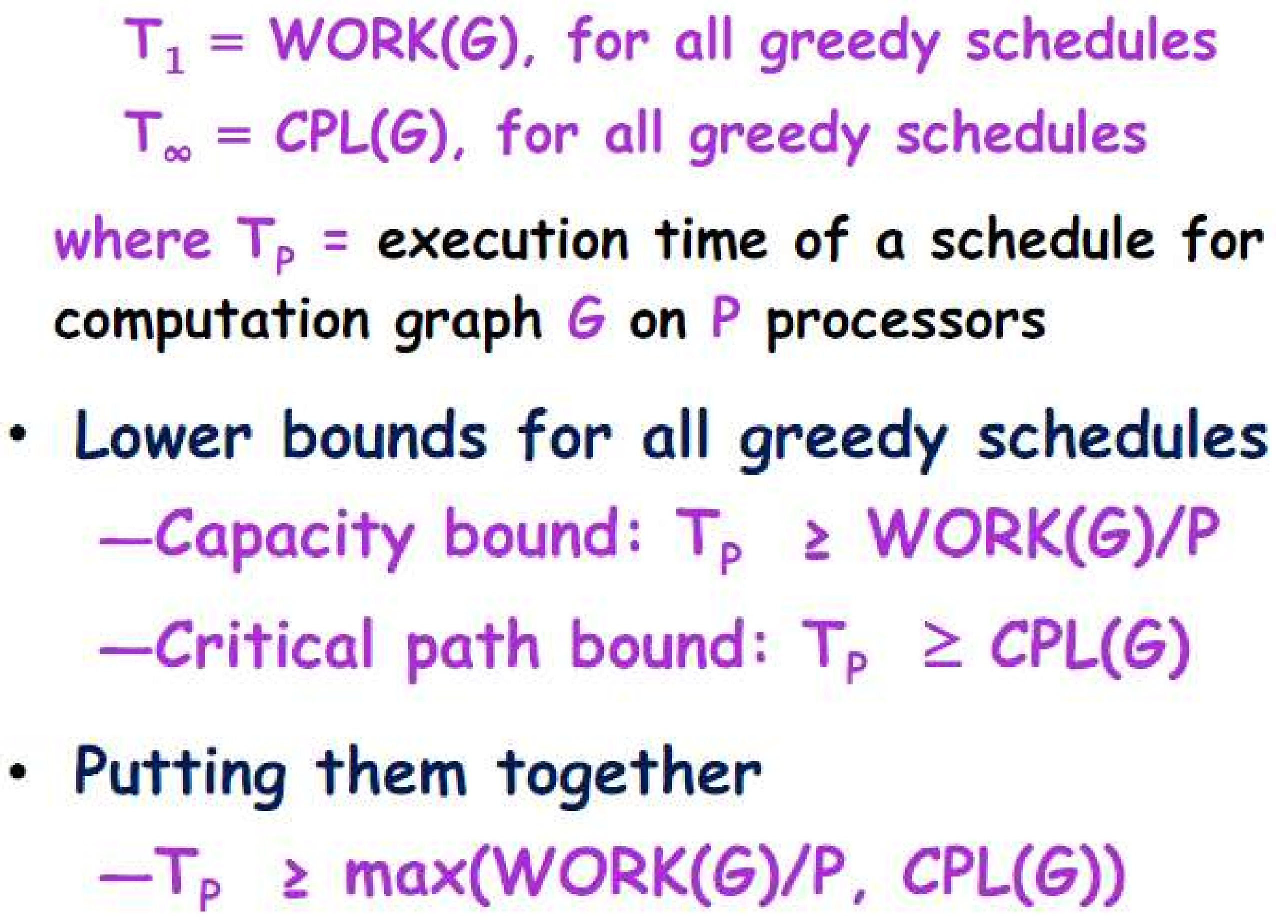
Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11	Completion time = 11		

NOTE: this schedule achieved a completion time of 11, which is the same as the CPL. Can we do better?

Greedy schedule

[https://wiki.rice.edu/confluence/display/PARPROG/COMP322 , lecture 3]

Def.: A greedy schedule never forces a processor
to be idle when at least one node is ready for execution.



T_1 = WORK(G), for all greedy schedules

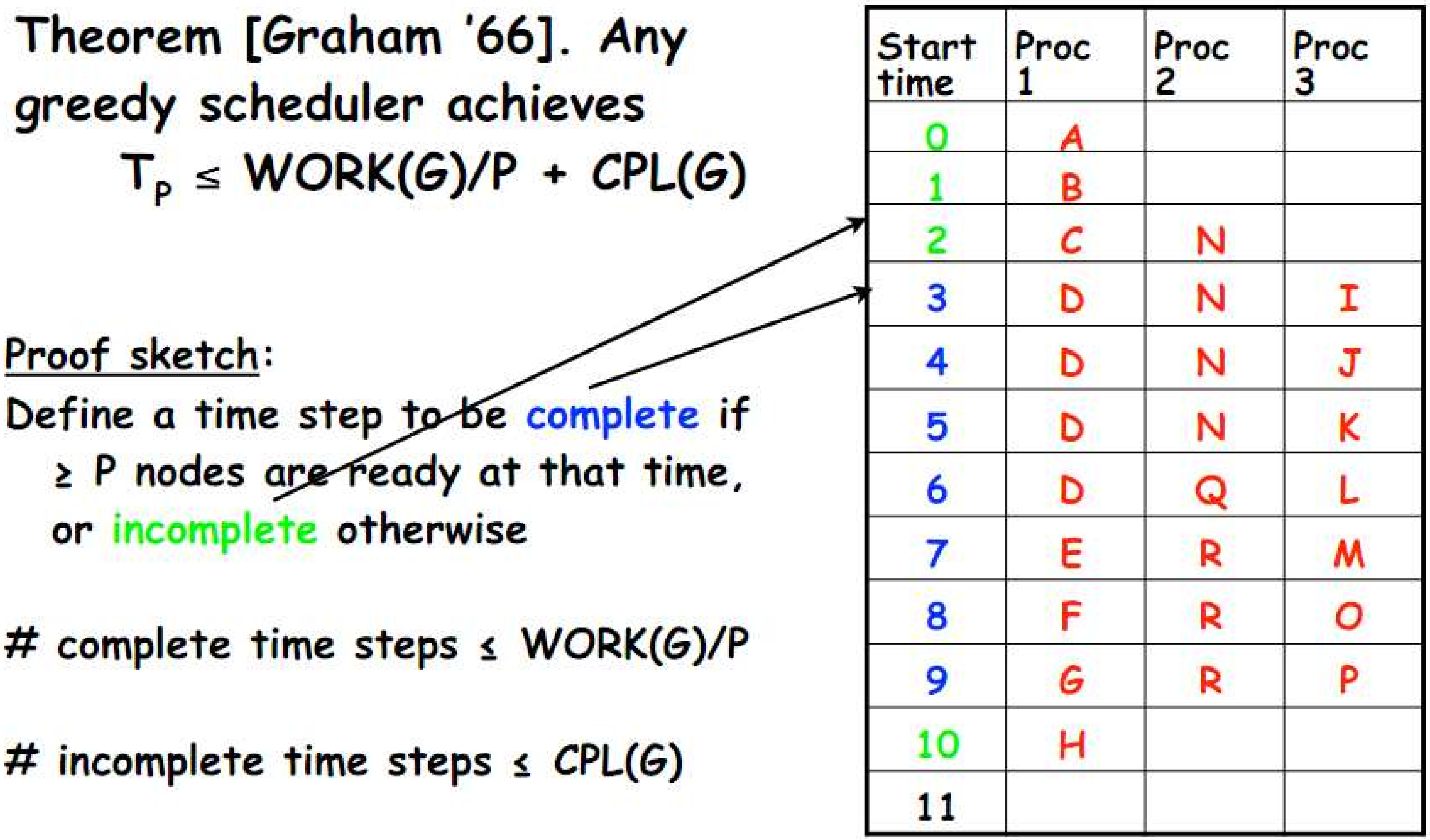
T_∞ = CPL(G), for all greedy schedules

where T_P = execution time of a schedule for computation graph G on P processors

- Lower bounds for all greedy schedules
  - Capacity bound: $T_P \geq$ WORK(G)/P
  - Critical path bound: $T_P \geq$ CPL(G)
- Putting them together
  - $T_P \geq$ max(WORK(G)/P, CPL(G))

Upper bound for greedy schedules

[https://wiki.rice.edu/confluence/display/PARPROG/COMP322 , lecture 3]



Theorem [Graham '66]. Any greedy scheduler achieves

$$T_P \leq WORK(G)/P + CPL(G)$$

Proof sketch:

Define a time step to be complete if ≥ P nodes are ready at that time, or incomplete otherwise

complete time steps ≤ WORK(G)/P

incomplete time steps ≤ CPL(G)

Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11			

Conclusion: Any greedy scheduler achieves T_p at most twice longer than the optimal time!

Using threads – examples

Server applications can handle multiple clients by launching a thread to deal with each client.

Long computations or high-latency disk and network operations can be handled in the background without disturbing foreground computations or screen updates.

Time measurement (via a timer) can be done in a separate thread.

Editing a source code while compiling (the previous version of) the code at the same time.

Using threads – examples, cont’d

App: web browser.

[http://www.cs.fiu.edu/~weiss/cop3338_f00/lectures/Threads/threads.pdf]

- separate thread downloads each image on a page (could be one thread per image),
- separate thread displays HTML,
- separate thread allows typing or pressing the stop button,
- makes browser look more responsive.

Why threads

- simpler programs (hm…)
- high degree of control
- (possibly) faster execution
- making use of multiple cores/CPUs.

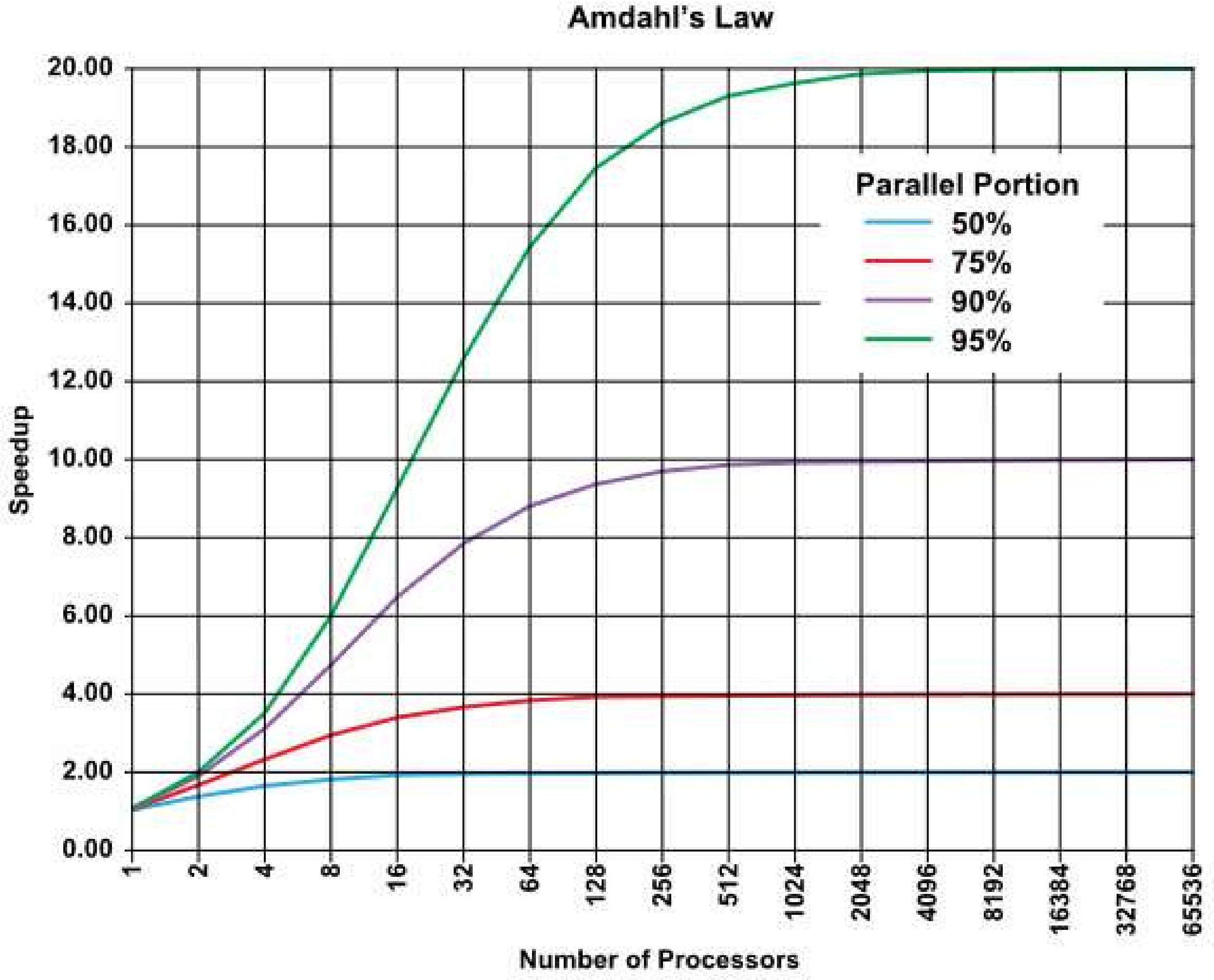
Beware of Amdahl’s law!

[http://en.wikipedia.org/wiki/Amdahl's_law]

Max speedup:

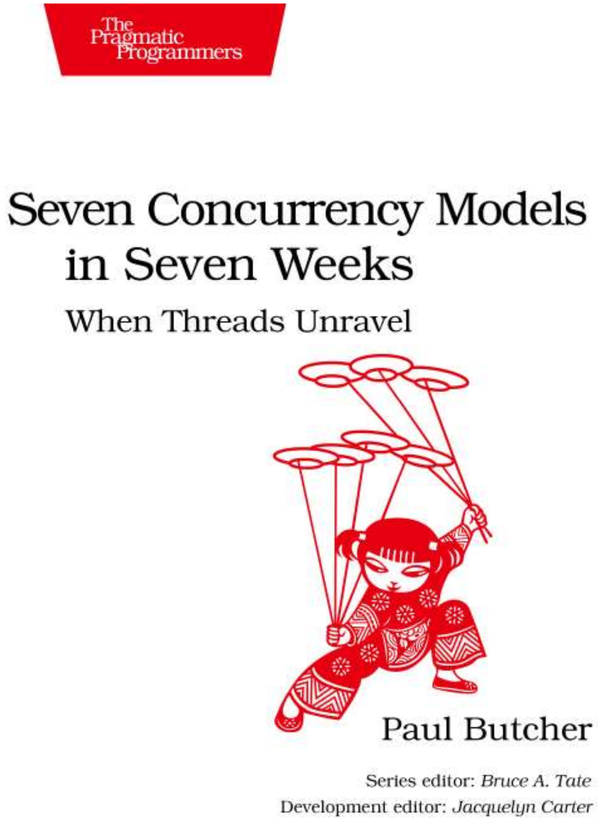
$$\frac{1}{(1-P)+\frac{P}{N}}$$

P – parallel component
S – serial component
(= 1 – *P*)
N – no of processors



Various concurrency models

- threads and locks,
- functional programming
  (pure functions, no mutable state),
- actors,
- communicating sequential processes,
- data parallelism (e.g., SIMD),
- …



The Pragmatic Programmers

Seven Concurrency Models in Seven Weeks

When Threads Unravel

Paul Butcher

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Concurrency and parallelism

A concurrent program has multiple logical threads of control.
These threads may or may not run in parallel.

A parallel program (usually) runs faster than a sequential program by executing different parts of the computation simultaneously (in parallel).
It may or may not have more than one logical thread of control.

We can also say that
concurrency is an aspect of the problem domain
(your program needs to handle multiple simultaneous,
or near-simultaneous, events).
Parallelism, by contrast, is an aspect of the solution domain
(you want to make your program faster by processing different portions of the problem in parallel).

Based on [P. Butcher, *Seven Concurrency Models in Seven Weeks*], chap. 1

Threads in the VM

- VM has threads in background
- VM alive as long as a “legitimate thread” still around (illegitimate threads are “daemons”)
- GUI programs will start separate thread to handle events once frame is visible

Spawning a thread (step-by-step)

1. Create your class, making it implement Runnable

```
class Animation implements Runnable { … }
```

2. Define a method in your class with the signature:

```
        public void run() {  // this method will be called
                // when the thread starts up
                …
                // when this exits, thread dies
        }
```

3. Create an instance of your class:

```
Animation a = new Animation();
```

4. Create a new thread attached to your new instance:

```
Thread t = new Thread(a);
```

5. Start the thread attached to your object: t.start();

Simplest example (1/2)

```
class BlahBlah implements Runnable {
        String str;
        public BlahBlah(String s) { str = s; }
        public void run() {
                while (true) {  // INFINITE LOOP!
                        System.out.print(str);
                }
        }
}
```

Simplest example (2/2)

Now we'll use BlahBlah:

```
class BlahBlahTest {
        public static void main(String[] args) {
                BlahBlah j = new BlahBlah("Oi! ");
                BlahBlah k = new BlahBlah("Yo! ");
                Thread t = new Thread(j);
                Thread u = new Thread(k);
                t.start();
                u.start();
        }
}
```

Output:

```
Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi!
Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo!
Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo!
Yo! Yo! Yo! Yo! Yo! Yo! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi!
Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi! Oi!
Oi! Yo! Yo! Yo! Yo! Yo! Yo! Yo! Yo!
```

Improving the output...

```
class BlahBlah implements Runnable {
        String str;
        public BlahBlah(String s) { str = s; }
        public void run() {
                while (true) {  // INFINITE LOOP!
                        System.out.print(str);
                        try { Thread.sleep(1); }
                        catch( InterruptedException e ) {}
                }
        }
}
....
```

Output:

```
Oi! Yo! Oi! Yo! Oi! Oi! Yo! Oi! Yo! Oi! Yo! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo!
Oi! Yo! Oi! Oi! Yo! Oi! Yo! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo!
Oi! Oi! Yo! Oi! Yo! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Oi! Yo! Oi!
Yo! Oi! Yo! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Yo! Oi! Oi! Yo! Oi! Yo! Yo!
```

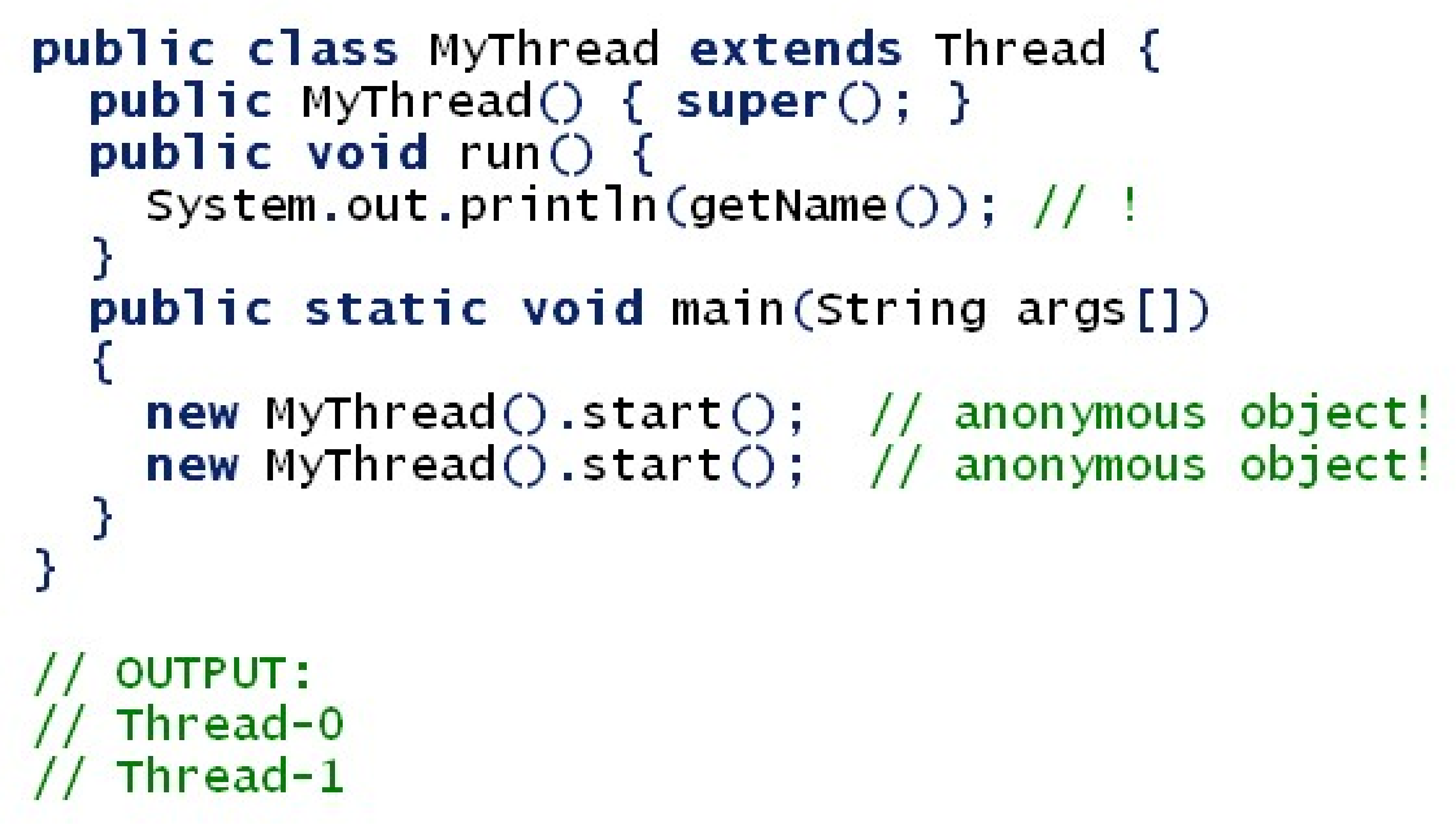
Spawning a thread, another way (1/2)

```
// Create your class extending Thread
import java.awt.*;
class Timer extends Thread {
      Label t1;
      Timer(Label l){ t1 = l; }

   // Redefine method run():
   public void run() {
      int time = 0;
      while (true) {
            try { this.sleep(500); }
            catch(InterruptedException exc){ return; }
            time++;
            t1.setText(String.valueOf(time));
      }
   }
}
```

Spawning a thread, another way (2/2)

```
// create an object of your class
public class Timer1 {
  public static void main(String[] args) {
      Frame f = new Frame("Threads");
      Label l = new Label("00000");
      l.setFont(new Font("Dialog", Font.BOLD, 44));
      f.add(l, "North");
      f.setSize(300,200);
      f.show();

      // send the start() message to your object
      Timer tm = new Timer(l); // here we create object
      tm.start();  // here we launch a thread
                   // counting half-secs
   }
}
```

We can name a thread, and get its name



```
public class MyThread extends Thread {
  public MyThread() { super(); }
  public void run() {
    System.out.println(getName()); // !
  }
  public static void main(String args[])
  {
    new MyThread().start();  // anonymous object!
    new MyThread().start();  // anonymous object!
  }
}

// OUTPUT:
// Thread-0
// Thread-1
```

Analogously:
setName(String name)

States of a thread

[based on http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf]

A thread may be in one of the following states:

- New: the Thread object has been created using new, but start() has not yet been called for it.

- Runnable: It is able to run, but currently some other thread is being executed.

Some authors don't distinguish between Runnable and Running.

- Running: it is currently running, i.e. executing instructions on the computer (start() has been called).

- Blocked: The thread cannot proceed until something happens. For example, it might have called sleep() (or it might have called wait() – elaborated later). Or is blocking on I/O.

- Dead: The run() method has terminated. Cannot be "revitalized". That is, you can't restart it by calling start() again (i.e. no effect).

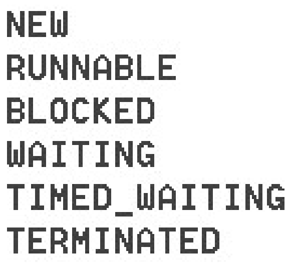
States of a thread, cont’d

([http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf],
[http://www.jguru.com/faq/view.jsp?EID=1253344])

We can find out some information about the state of a thread at any time, e.g.: **if (ball.isAlive()) { ... }**

The method **isAlive()** returns **true** if the thread is running, runnable or blocked, and returns **false** if it is new or dead.

Until Java 1.5, there was no way to distinguish between a new thread and a dead thread (unless the object deliberately sets a variable to say that it has finished running)!
Now we can write:

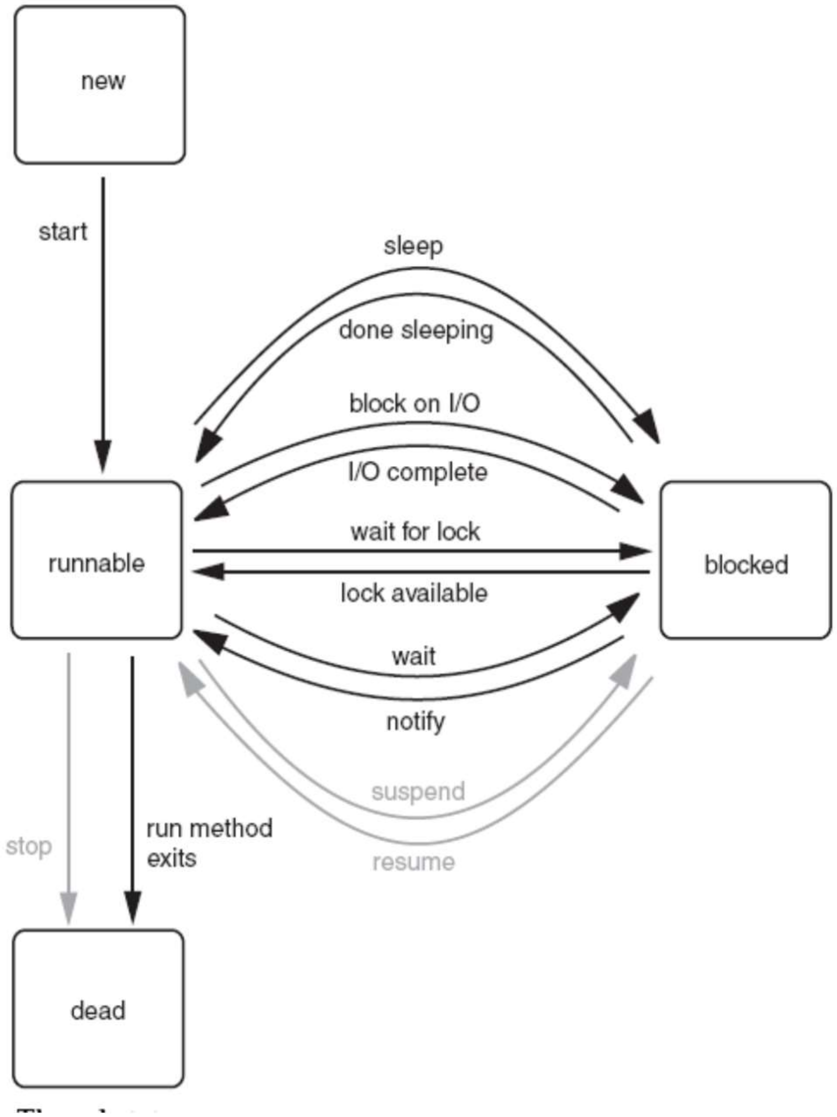


```
NEW
RUNNABLE
BLOCKED
WAITING
TIMED_WAITING
TERMINATED
```

```
// t – some thread
Thread.State e = t.getState(); // returns an enum
System.out.println(e.name()); // prints the current state of t
Thread.State[] ts = e.values();
for(int i = 0; i < ts.length; i++) { System.out.println(ts[i]); }
```

Thread lifecycle

[http://horstmann.com/corejava/cj7v2ch1.pdf]



Light-shaded methods (suspend, resume, stop) – deprecated!

Scheduling

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf]

Parallel thread execution on a single CPU core is an illusion, of course. At a time, only one thread is running.

How does the system decide which thread runs?
That is the job of the scheduler.

We cannot make any assumptions about how the scheduler works, as different strategies are used in different operating systems, or different versions of the Java Virtual Machine.

A thread executes only when it is in state *running*.

Invoking **start()** does NOT cause a thread to run (it merely becomes runnable, and must wait for the scheduler to give it a turn).

Daemon threads

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf]

```
t.setDeamon(true);  // set t as a daemon
// This method must be called before the thread is started.
```

Daemons are threads that are constantly running (once created), and typically perform some services. (like the garbage collection thread).

When only daemon threads remain, the VM exits.

Daemon example: a timer thread that sends regular “time ticks” to other threads.

Method *sleep(long time_in_millis)* is very useful

Static method (invoke: *Thread.sleep(...);*).

Current thread gives up the processor for at least the time specified.

Time in millisec.

No guarantee that you get processor back.

Must catch *InterruptedException*.

Also a form sleep(long millis, int nanos) exists.

Other static methods from Thread

static Thread currentThread() –
returns the object representing the currently running thread

static int enumerate(Thread[] threadArray) –
copies into the specified array every active
thread in the current thread's thread group and its subgroups. Returns the
number of threads put into the array

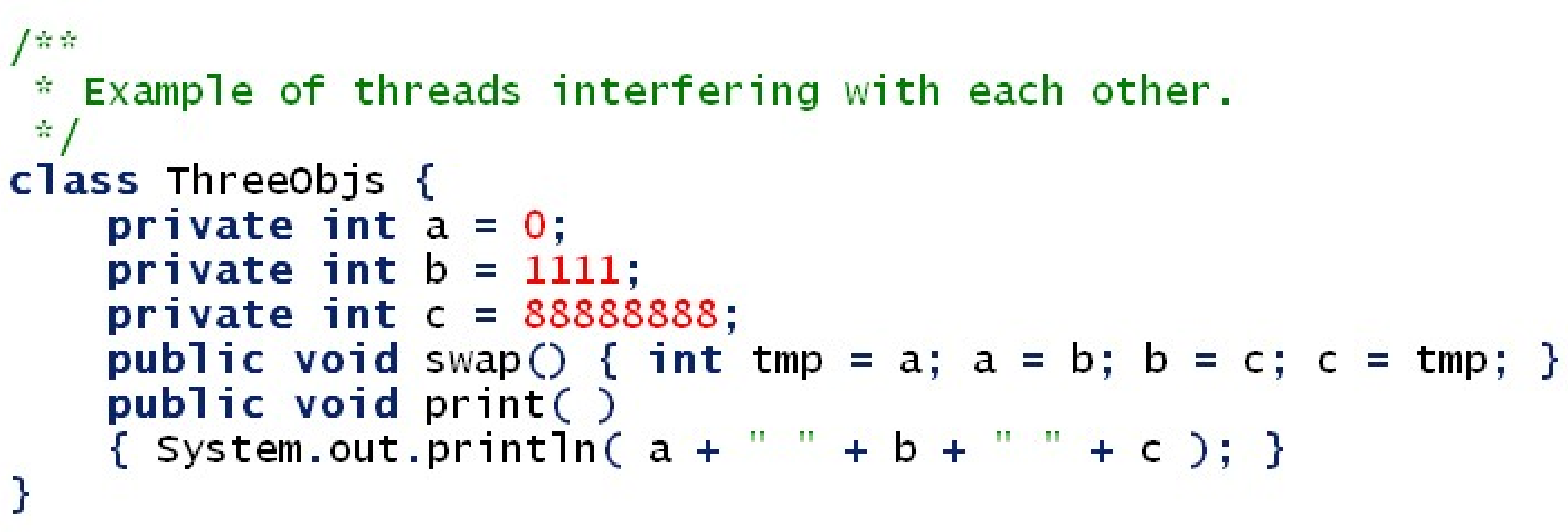
static int activeCount() –
returns the number of active threads in the
current thread's thread group

J. Bloch, *Effective Java*, item 73: *(…) thread groups are obsolete. (…) If you design a class that deals with logical groups of threads, you should probably use thread pool executors.*

Houston, we got a problem!

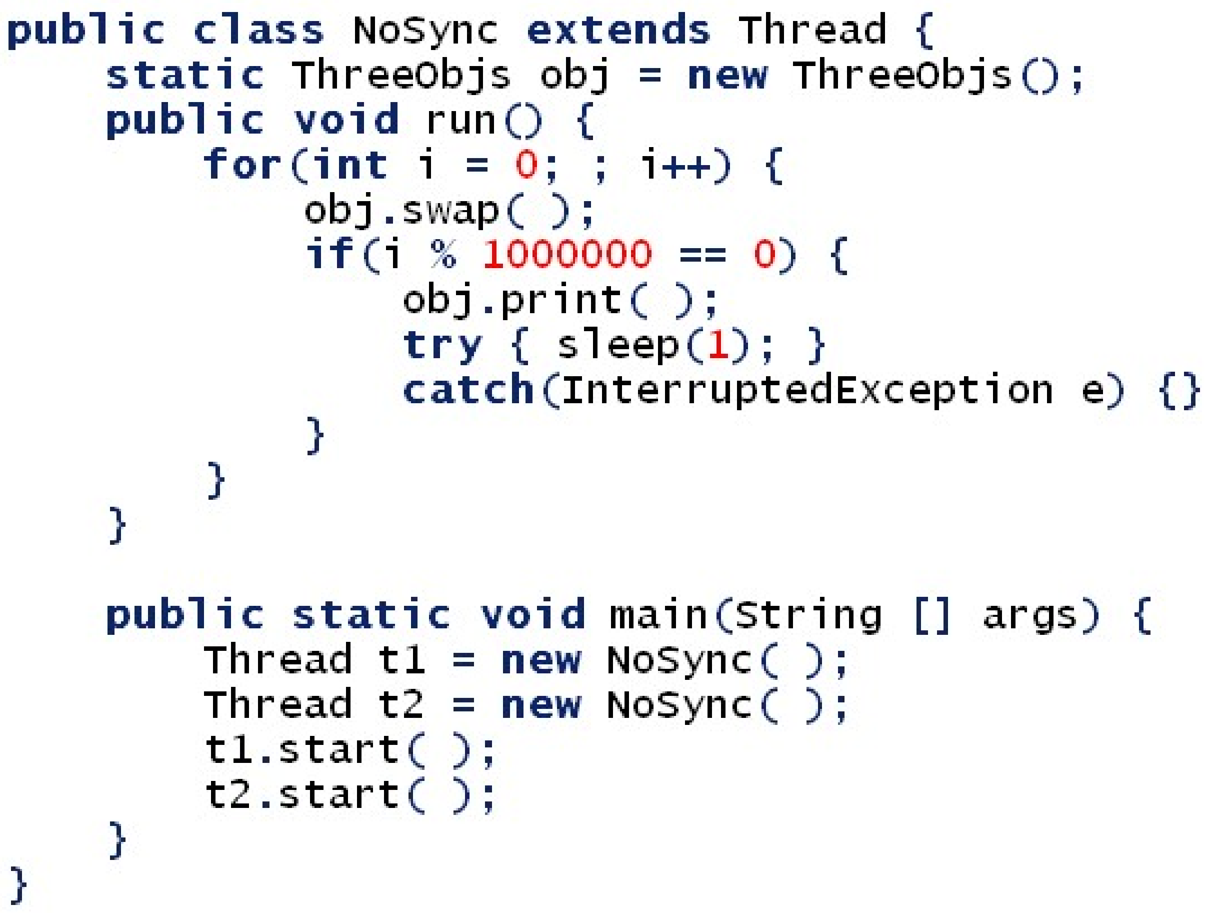
[http://www.cs.fiu.edu/~weiss/cop3338_f00/lectures/Threads/NoSync.java]

A little example showing that threads may disturb to each other.



```
/**
 * Example of threads interfering with each other.
 */
class ThreeObjs {
    private int a = 0;
    private int b = 1111;
    private int c = 88888888;
    public void swap() { int tmp = a; a = b; b = c; c = tmp; }
    public void print( )
    { System.out.println( a + " " + b + " " + c ); }
}
```

NoSync.java, cont'd



```java
public class NoSync extends Thread {
    static ThreeObjs obj = new ThreeObjs();
    public void run() {
        for(int i = 0; ; i++) {
            obj.swap( );
            if(i % 1000000 == 0) {
                obj.print( );
                try { sleep(1); }
                catch(InterruptedException e) {}
            }
        }
    }

    public static void main(String [] args) {
        Thread t1 = new NoSync( );
        Thread t2 = new NoSync( );
        t1.start( );
        t2.start( );
    }
}
```

Output (smth like that...):

```
1111 88888888 0
88888888 0 1111
1111 88888888 0
1111 88888888 0
88888888 0 1111
1111 88888888 0
1111 88888888 0
88888888 0 1111
0 1111 88888888
88888888 0 1111
0 1111 88888888
88888888 0 1111
1111 88888888 0
88888888 0 1111
0 1111 88888888
1111 88888888 0
1111 88888888 0
0 1111 88888888
0 1111 88888888
```

NoSync.java output, cont’d

```
.....
1111 88888888 0
0 0 88888888
0 88888888 0
0 0 88888888
0 0 88888888
0 88888888 0
.....
```

Seen that?!
1111 disappeared!

```
.....
0 88888888 88888888
0 88888888 88888888
88888888 88888888 88888888
88888888 88888888 88888888
88888888 88888888 88888888
88888888 88888888 88888888
.....
```

Got stuck with 88888888’s.

(In another run on my machine (WinXP, Java 1.5.0) it was all 0’s...)

Shared data, problem gradation

Two threads access the same object, but only read data: no problem.

Two threads access the same object, but only one of them changes data: no good, as the object may be temporarily in a bad state.

Example [http://www.cs.fiu.edu/~weiss/cop3338_f00/lectures/Threads/threads.pdf] :

```
class TwoObjs {
  private int a = 15;   private int b = 37;
  public int sum( ) { return a + b; } // should always be 52
  public void swap( ) { int tmp = a; a = b; b = tmp; }
}
// assume time-slice in swap() ends after a=b;
// the other thread calls sum()... ☹
```

Two threads access the same object, and both change the object: fatal (we’ve just seen it, NoSync.java).

Thread interference, another example

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf]

Suppose that we have two threads:

– one sets new dates, using **setDate**,

– the other makes periodic calls of **getDate**.

Suppose now we have the date set to March, 31.

...And one thread makes the call

theDate.setDate("April", 1);

Let there be the month first set...

Thread interference, example, cont’d

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf]

...and just now – surprise, surprise – the pre-emptive scheduler decides (after only executing the first line in **setDate**) that this thread’s time-slot is over and that another thread is to be run.

Say this other different thread makes the call
String current = theDate.getDate();

What will be the value of current?

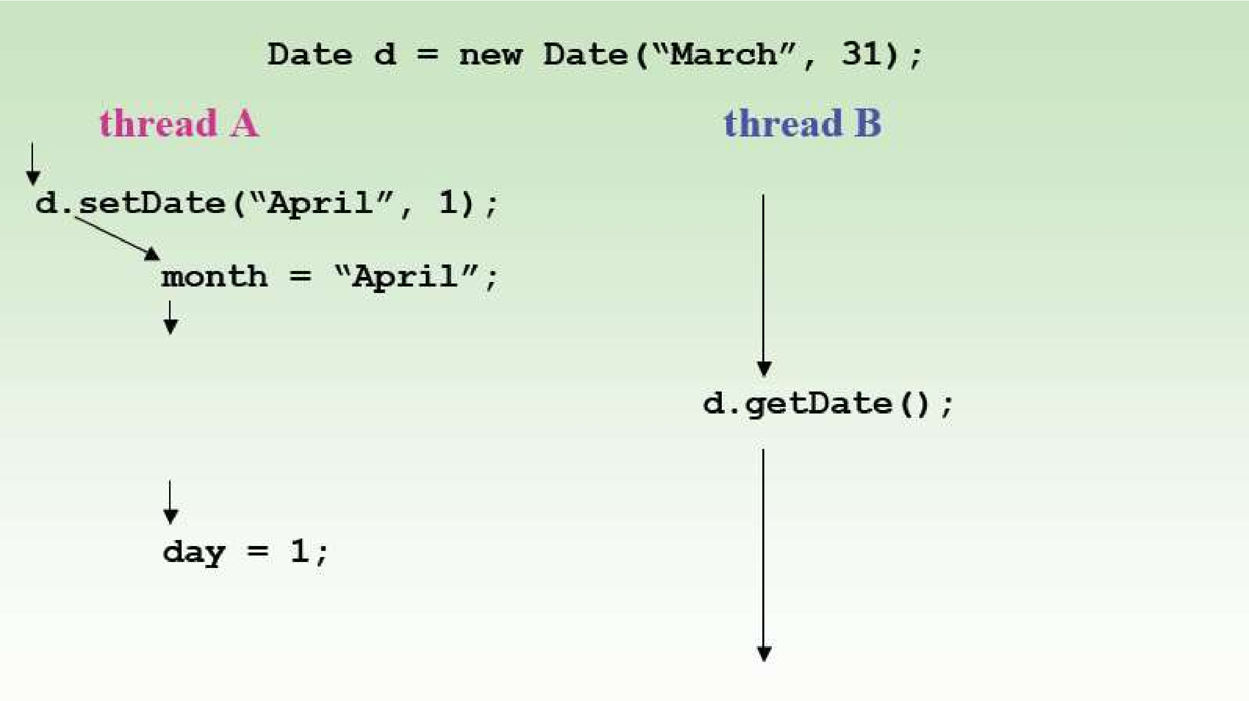
"April, 31". Of course. ☹

Thread interference, example, cont’d

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf]

You might say that it’s very unlikely that the execution of the thread will be suspended exactly between execution of those two assignment statements.

This is true. THAT’S EVEN WORSE, as this means that testing multi-threading programs is damn hard.



```
Date d = new Date(“March”, 31);
```

thread A

```
d.setDate(“April”, 1);
    month = “April”;
    day = 1;
```

thread B

```
d.getDate();
```

Some terms in concurrent programming

- Mutual Exclusion – one must be certain that concurrent processes have mutual exclusive access to shared data.
- Race Condition – one must avoid situations in which the relative speed of various processes affect the outcome.
- Deadlock & Starvation – one must avoid situations in which processes are placed in a locked condition indefinitely.

How to stop threads from interfering with each other [http://www.cs.stir.ac.uk/courses/31V4/lectures/java7.pdf]

Threads may well need to communicate information to each other.

- The simplest way of achieving this is to allow several threads to access the same stored information. This leads to the idea of shared variables.

- In the date example, there was one shared variable: the date object.

- We had a typical problem of what happens if threads try to share variables: one thread tries to update a shared variable at the same time that another thread is reading it.

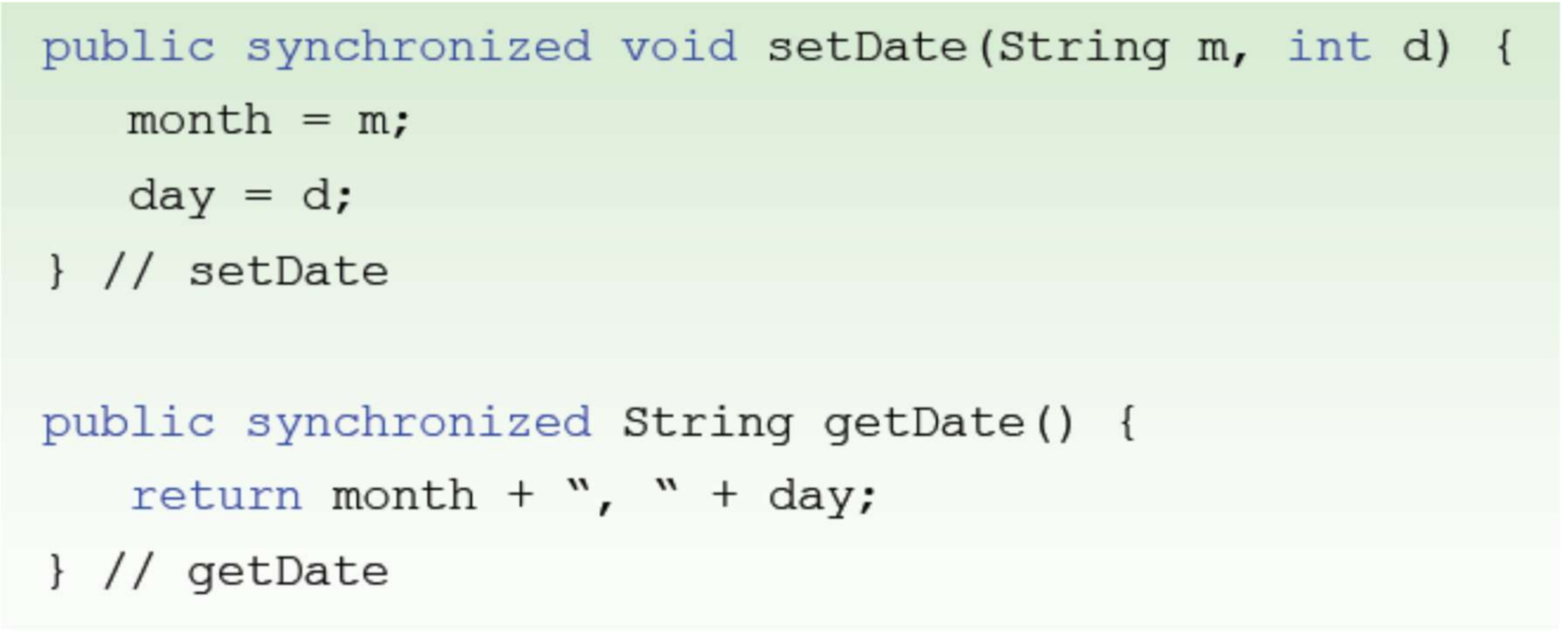
- This can lead to inconsistent information being read.

Critical regions

- A solution: the idea of having critical regions in our program.
- All access to shared variables takes place within these critical regions, and Java uses monitors to control access to the critical regions.
- For example, the setDate and getDate methods might form a critical region.
- Java provides mechanisms so that we can be confident that only one thread can be executing (whether currently suspended by the scheduler or not) within a critical region at any one time.
- If a second thread wishes to access to the critical region, it needs to be blocked until the first thread has left.
- This is mutual exclusion, i.e. threads have mutually exclusive access to the shared variables.

Keyword *synchronized*

We achieve all this in Java through synchronized methods.
Our Date class would be the same as before except for the word **synchronized**
being inserted in the relevant method headers:



```
public synchronized void setDate(String m, int d) {
   month = m;
   day = d;
} // setDate

public synchronized String getDate() {
   return month + “, ” + day;
} // getDate
```

Synchronizing a block

Sometimes no need to synchronize the entire method.

Just need to synchronize a “critical section” – few lines of code that should be viewed as an “atomic” single operation.

Use a synchronized block then:

```
synchronized( anyobject ) { ... }
```

As long as a given thread is inside a synchronized method or a synchronized block, any other thread attempting to invoke this or another method of the synchronized object must wait until the first thread leaves the synchronized method (block).

Version #1 and #2 are equivalent

```
public class C1 {  // Version #1
  synchronized public void f1() { ... }
  synchronized static void f2() { ... }
}

public class C1 {  // Version #2
  public void f1() {
          synchronized(this) { ... }
  }
  static void f2() {
          synchronized(C1.class) { ... }
  }
}
```

synchronized in method header is just a convenience. *Synchronized* is NOT inherited.

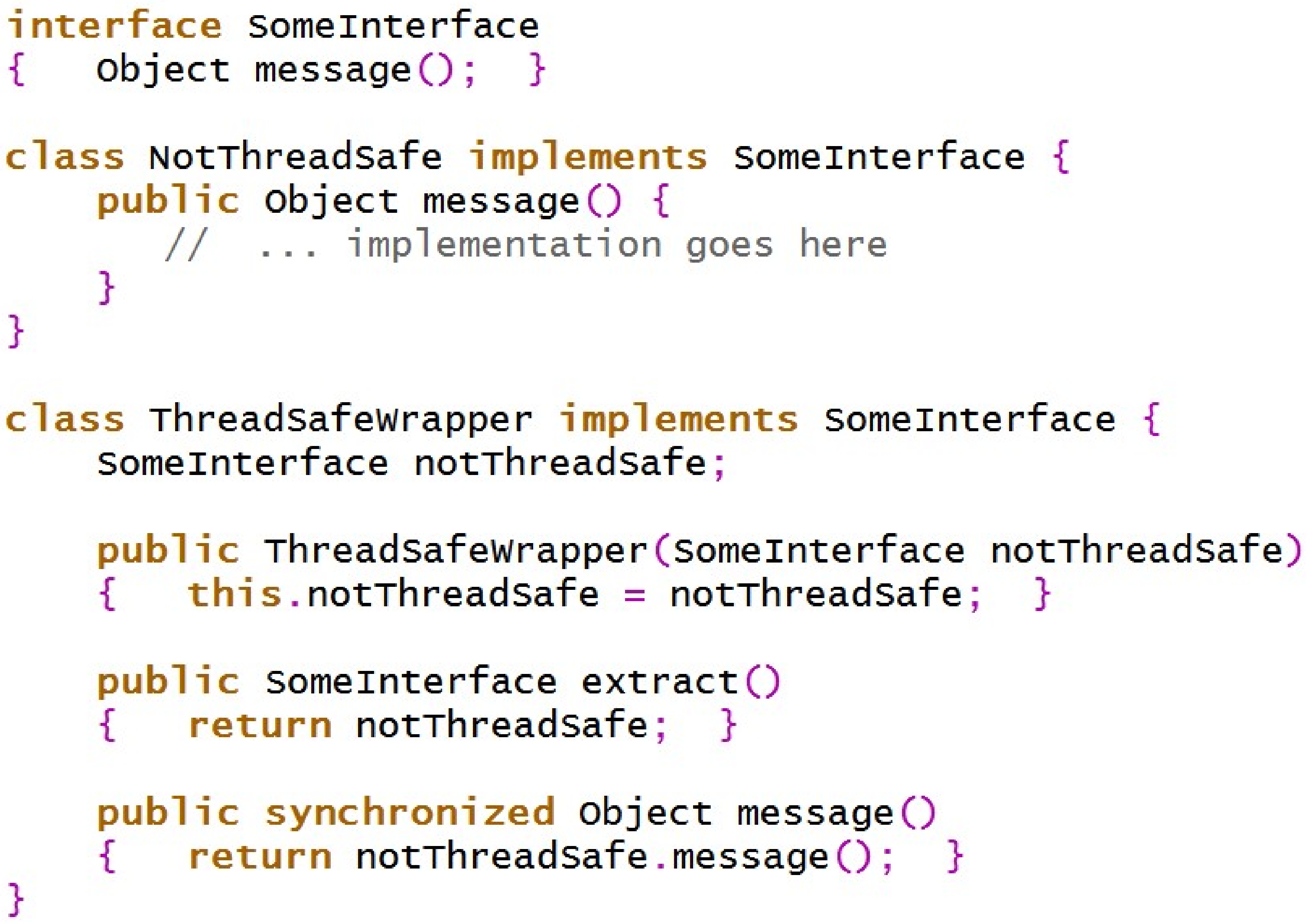
Synchronization

Synchronization

- prevents a thread from observing an object in an inconsistent state,
- ensures that each thread entering a synchronized method or block sees the effects of all previous modifications that were guarded by the same lock.

When multiple threads share mutable data, each thread that reads or writes the data must perform synchronization.

Synchronization wrappers. Example



```
interface SomeInterface
{   Object message();  }

class NotThreadSafe implements SomeInterface {
    public Object message() {
       //  ... implementation goes here
    }
}

class ThreadSafeWrapper implements SomeInterface {
    SomeInterface notThreadSafe;

    public ThreadSafeWrapper(SomeInterface notThreadSafe)
    {   this.notThreadSafe = notThreadSafe;  }

    public SomeInterface extract()
    {   return notThreadSafe;  }

    public synchronized Object message()
    {   return notThreadSafe.message();  }
}
```

Based on [A. Holub, *Taming Java Threads*, 2000]

Synchronization wrappers, cont’d

When thread safety isn’t an issue, you can just declare and use objects of class NotThreadSafe.

When you need a thread-safe version, just wrap it:

```
SomeInterface object = new NotThreadSafe();
// ...
object = new ThreadSafeWrapper(object);
// object is now thread-safe
```

When you don’t need thread-safe access any more, unwrap it:

```
object = ((ThreadSafeWrapper)object).extract();
```

Synchronization is (often) not enough

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java9.pdf]

Interacting Threads – Producer/Consumer

- In the setDate/getDate example, we do not care how often a date, once set, is accessed using getDate.
- But sometimes where we need to add the condition that the information must be read exactly once.
- This is known as the producer/consumer problem.
- A Producer produces items. A Consumer consumes items. E.g., Author / Writer example in [Barteczko'04]: Author creates stories, Writer "catches" them and prints (sends to the screen).

Producer / Consumer, cont’d

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java9.pdf]

- We must ensure that all items produced by Producer are consumed by Consumer, and that Consumer does not consume the same item twice.

Hence:

– if Producer is faster than Consumer, then Producer must wait for the item to be consumed, and

– if Consumer is faster than Producer, then Consumer must wait until the next item has been produced.

Solution?

1. Use a buffer (queue)?

In one thread, **Producer** produces items which it puts in the buffer, in the other thread, **Consumer** takes items from the buffer and “consumes” them.

The **Buffer** encapsulates the shared variable,
and is the critical region.

Producer and **Consumer** access the shared variable (which stores the items) through synchronized **give** and **take** methods from the **Buffer** class.

Gives mutual exclusion (because the methods are **synchronized**).

OK?

Solution? (cont'd)

WRONG. ☹ As well as mutual exclusion,
we need to ensure that an item can only be put in an empty buffer, and removed from a full buffer.

We are going to use wait() and notify() from class Thread

When a thread invokes **wait()**,
it is put in state blocked and loses the lock.

The effect of **notify()** is to wake up a thread which has executed a call of **wait()** in this object. If there is no waiting thread, then the call has no effect.

When a thread is woken up, it becomes runnable.
(But it is run again only when the scheduler decides.)

wait()

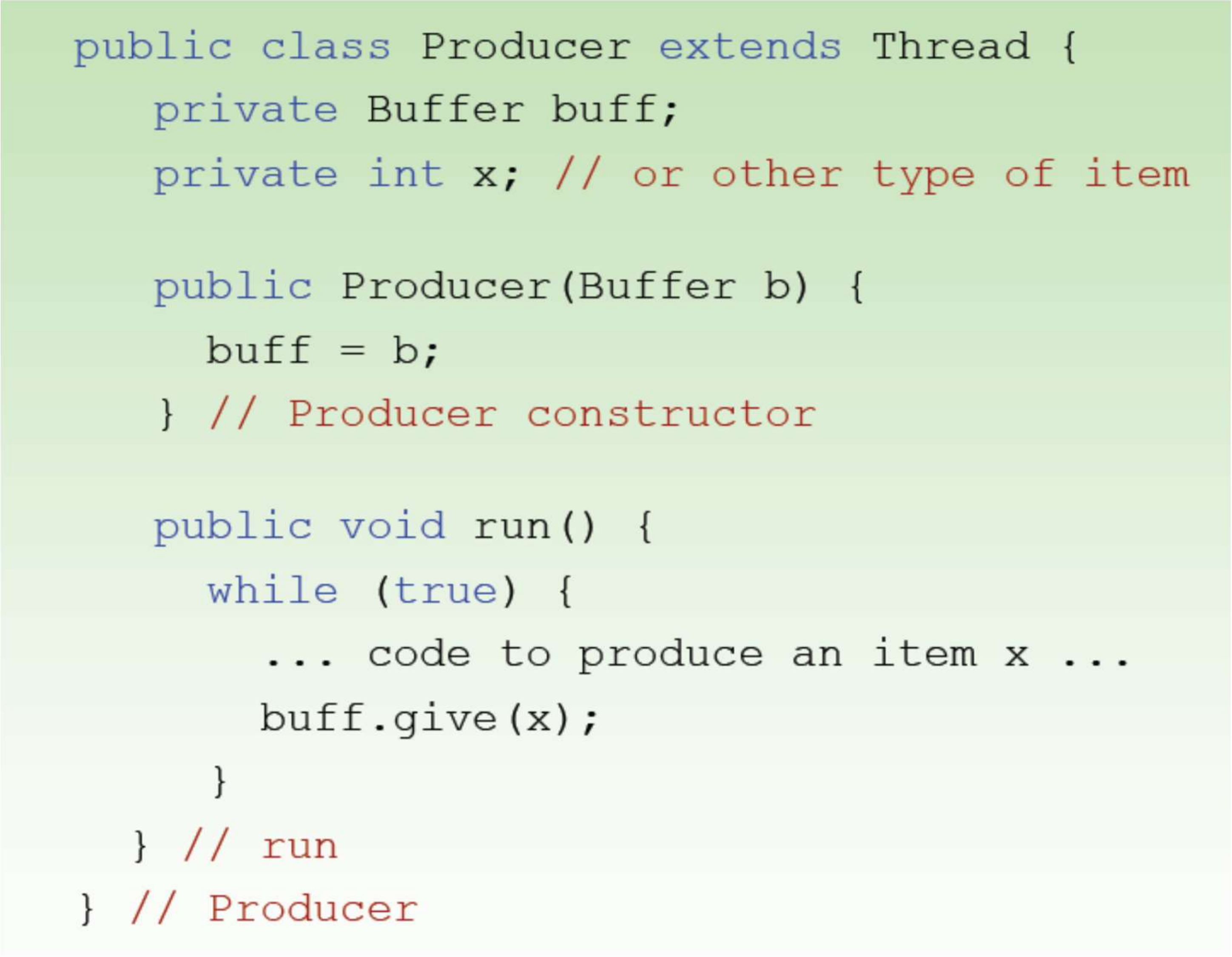
wait() must always occur within a **try/catch** block:

```
try {
        wait();
}
catch (InterruptedException e) {
        System.err.println("Exception");
}
```

Now, let’s present all the needed classes...

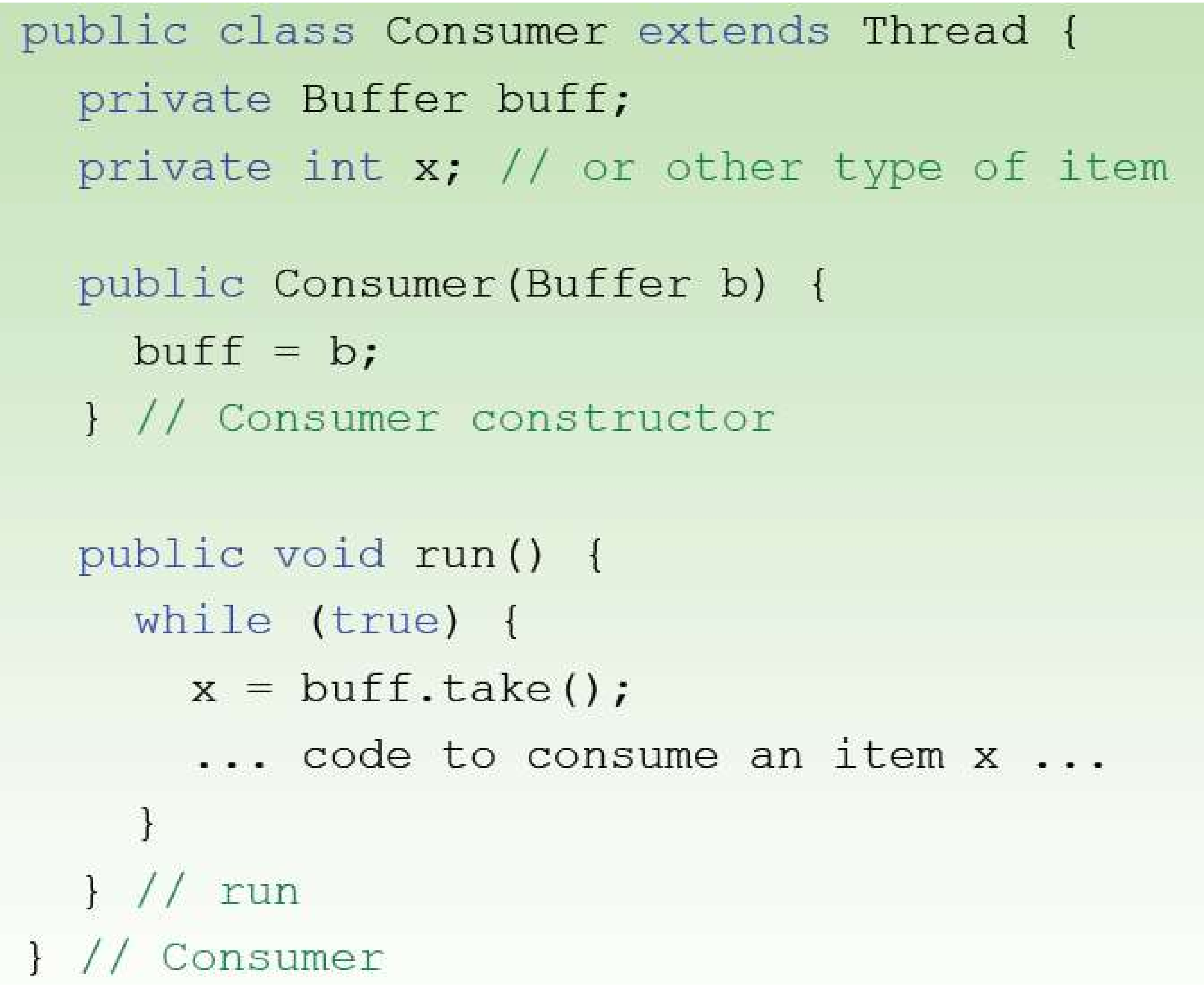
Many of them...
Producer, Consumer, Buffer and ProduceConsume.

Producer (produces items and puts them in a buffer)



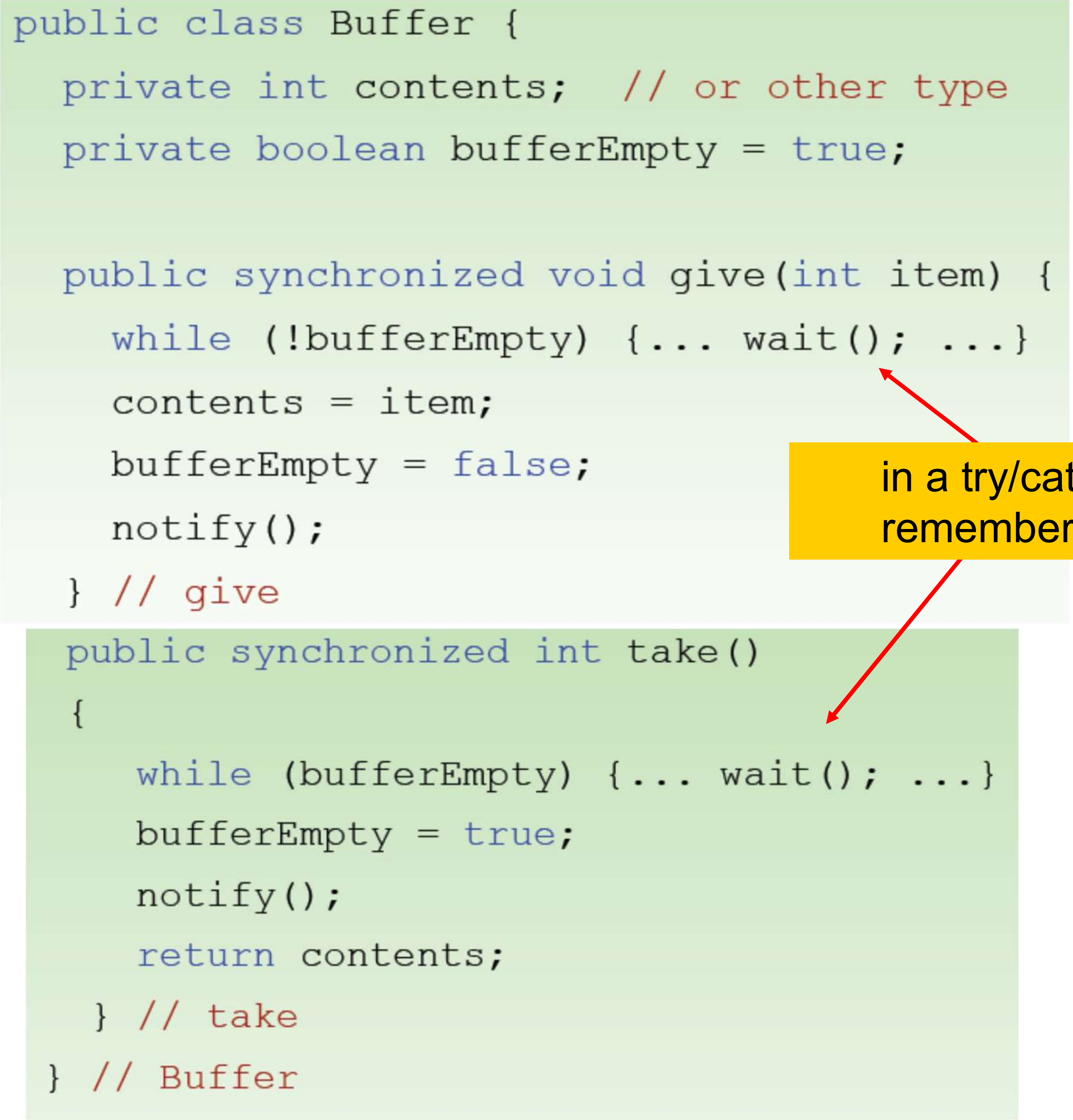
```java
public class Producer extends Thread {
   private Buffer buff;
   private int x; // or other type of item

   public Producer(Buffer b) {
     buff = b;
   } // Producer constructor

   public void run() {
     while (true) {
       ... code to produce an item x ...
       buff.give(x);
     }
  } // run
} // Producer
```

Consumer (takes items from a buffer)



```
public class Consumer extends Thread {
  private Buffer buff;
  private int x; // or other type of item

  public Consumer(Buffer b) {
    buff = b;
  } // Consumer constructor


  public void run() {
    while (true) {
      x = buff.take();
      ... code to consume an item x ...
    }
  } // run
} // Consumer
```

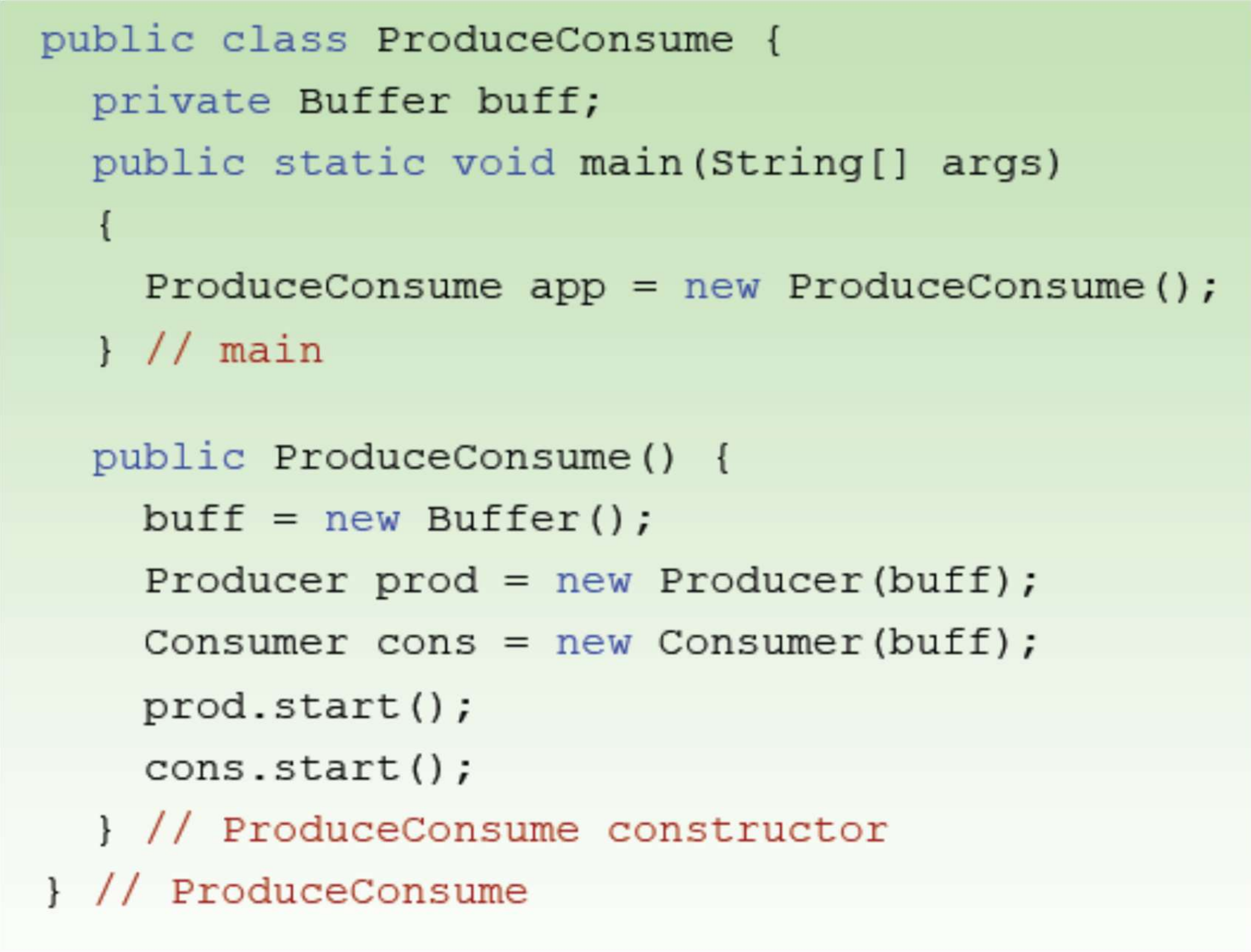
Buffer (critical region; only one thread may be executing one of its methods at any one time)



```
public class Buffer {
  private int contents;   // or other type
  private boolean bufferEmpty = true;

  public synchronized void give(int item) {
    while (!bufferEmpty) {... wait(); ...}
    contents = item;
    bufferEmpty = false;
    notify();
  } // give
 public synchronized int take()
 {
    while (bufferEmpty) {... wait(); ...}
    bufferEmpty = true;
    notify();
    return contents;
  } // take
} // Buffer
```

in a try/catch block, remember!

Putting it all together



```
public class ProduceConsume {
  private Buffer buff;
  public static void main(String[] args)
  {
    ProduceConsume app = new ProduceConsume();
  } // main

  public ProduceConsume() {
    buff = new Buffer();
    Producer prod = new Producer(buff);
    Consumer cons = new Consumer(buff);
    prod.start();
    cons.start();
  } // ProduceConsume constructor
} // ProduceConsume
```

Producer / Consumer – extension

- We could have several Producer and Consumer objects running in parallel in separate threads

– When a thread calls wait(), the thread is blocked.

– Several threads can be blocked at any one time.

- When a call of notify() is executed, one of the blocked threads is reactivated, i.e. put in state Runnable.

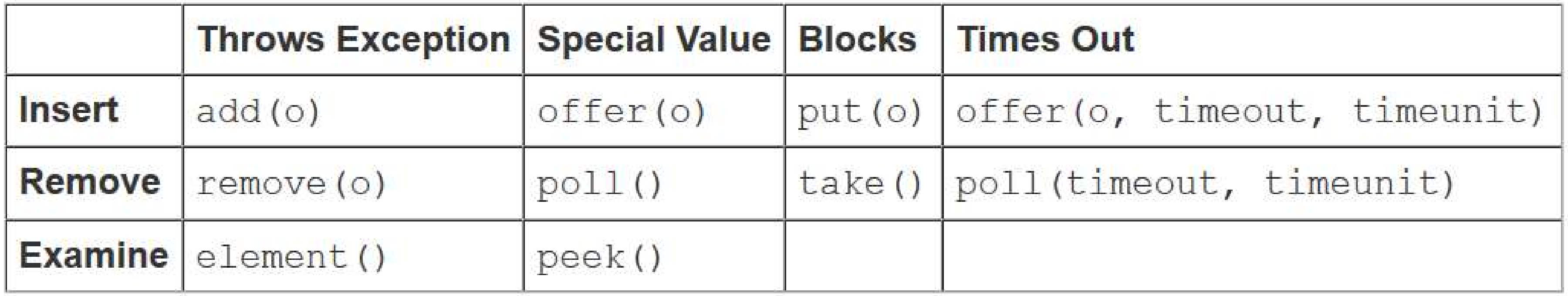
- For our example to work, a Producer needs to notify a Consumer, and a Consumer needs to notify a Producer...

– …but it's possible for a Producer to notify another Producer (or a Consumer to notify another Consumer).

- To get around this problem, there is a method notifyAll(). This can be used in place of notify(), and it wakes up all threads waiting on this object (i.e. sets them all to be runnable).

Simpler: use a queue implementing java.util.concurrent.BlockingQueue<E>

BlockingQueue methods:



	Throws Exception	Special Value	Blocks	Times Out
Insert	`add(o)`	`offer(o)`	`put(o)`	`offer(o, timeout, timeunit)`
Remove	`remove(o)`	`poll()`	`take()`	`poll(timeout, timeunit)`
Examine	`element()`	`peek()`		

Blocking methods:

put (if the producer cannot add an item to the queue, as it is full),

take (when the consumer cannot take an item since the queue is empty).

http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html

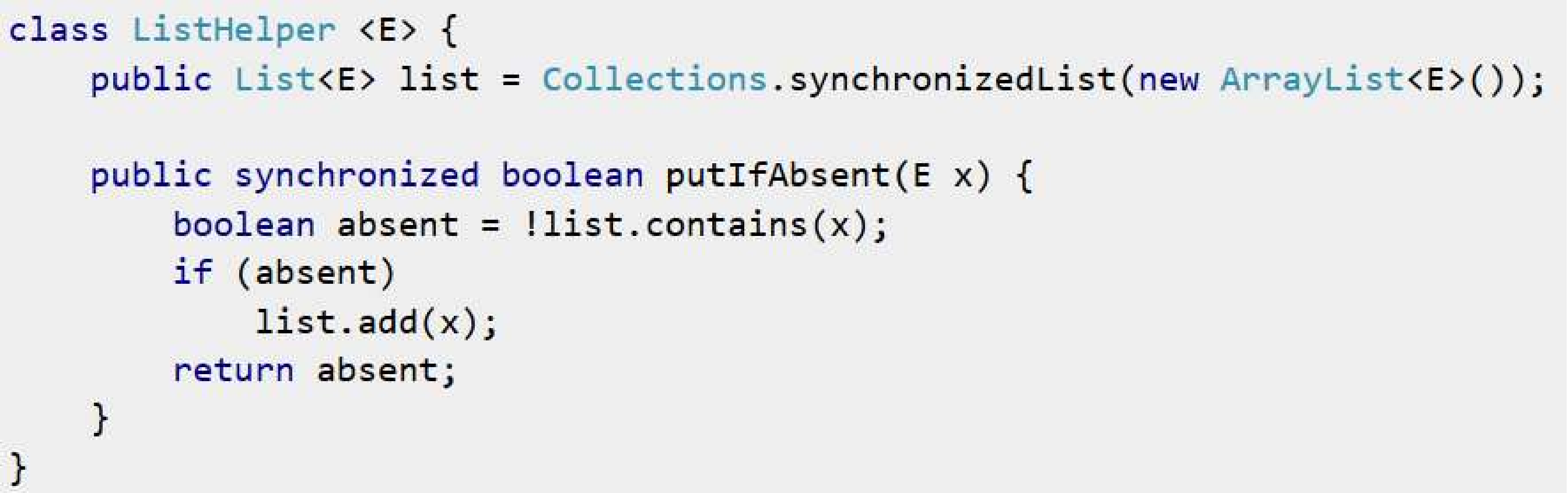
Recommended BlockingQueue implementations

SynchronousQueue: does not have any internal capacity! You cannot insert an element (using any method) unless another thread is trying to remove it.

ArrayBlockingQueue: backed by an array, its capacity is set in the constructor(s).

Careful with locking!

What’s wrong with this code? We have *synchronized* keyword…



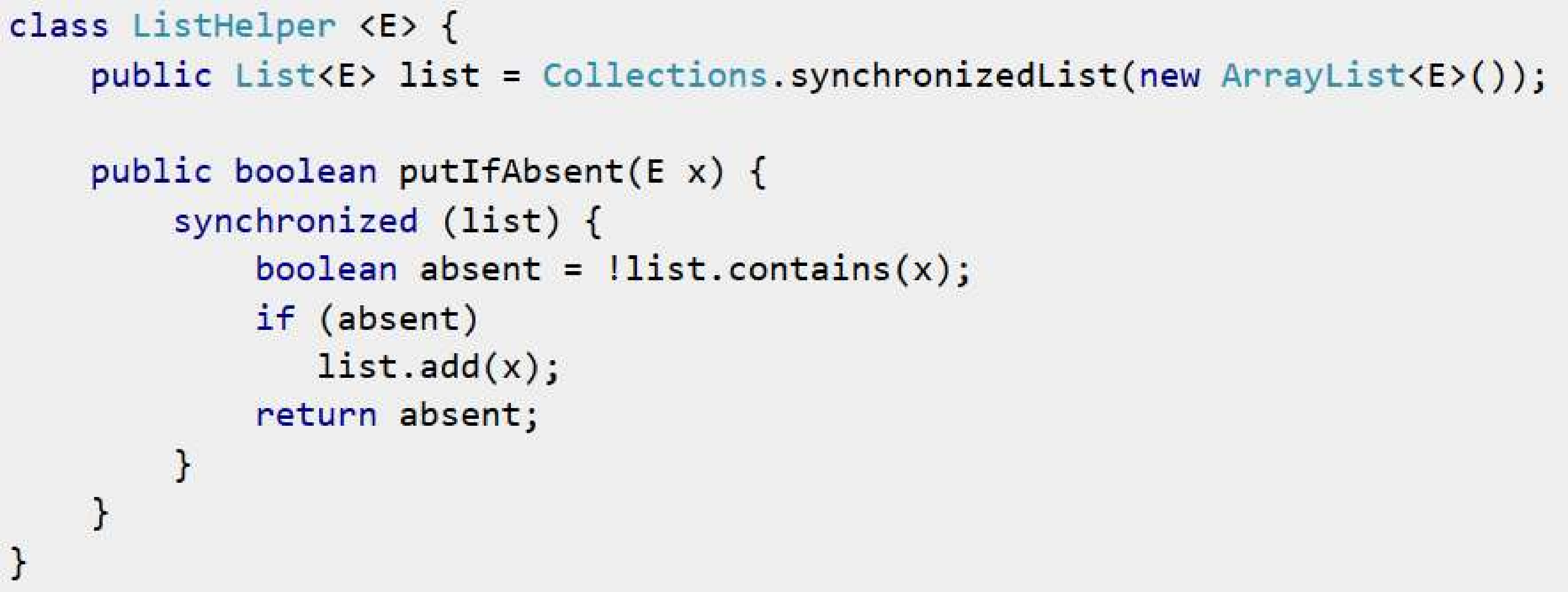
```
class ListHelper <E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
}
```

Yet, the list is unlocked when *contains* returns,
and then locked again when *add* is called.
Something else could add the same element between the two.

That’s because the method putIfAbsent synchronizes on a wrong lock. *this*, i.e. object of ListHelper, gives no guarantee that another thread won’t modify the list while putIfAbsent is executing.

http://stackoverflow.com/questions/17533263/why-is-this-code-not-thread-safe-even-when-using-a-synchronized-method

Fixed version



```
class ListHelper <E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public boolean putIfAbsent(E x) {
        synchronized (list) {
            boolean absent = !list.contains(x);
            if (absent)
               list.add(x);
            return absent;
        }
    }
}
```

Alternatively: make *list* private (then no other objects can access it).
Then also Collections.synchronizedList is not needed,
as you're providing all necessary synchronization in your own code.

Atomic operations

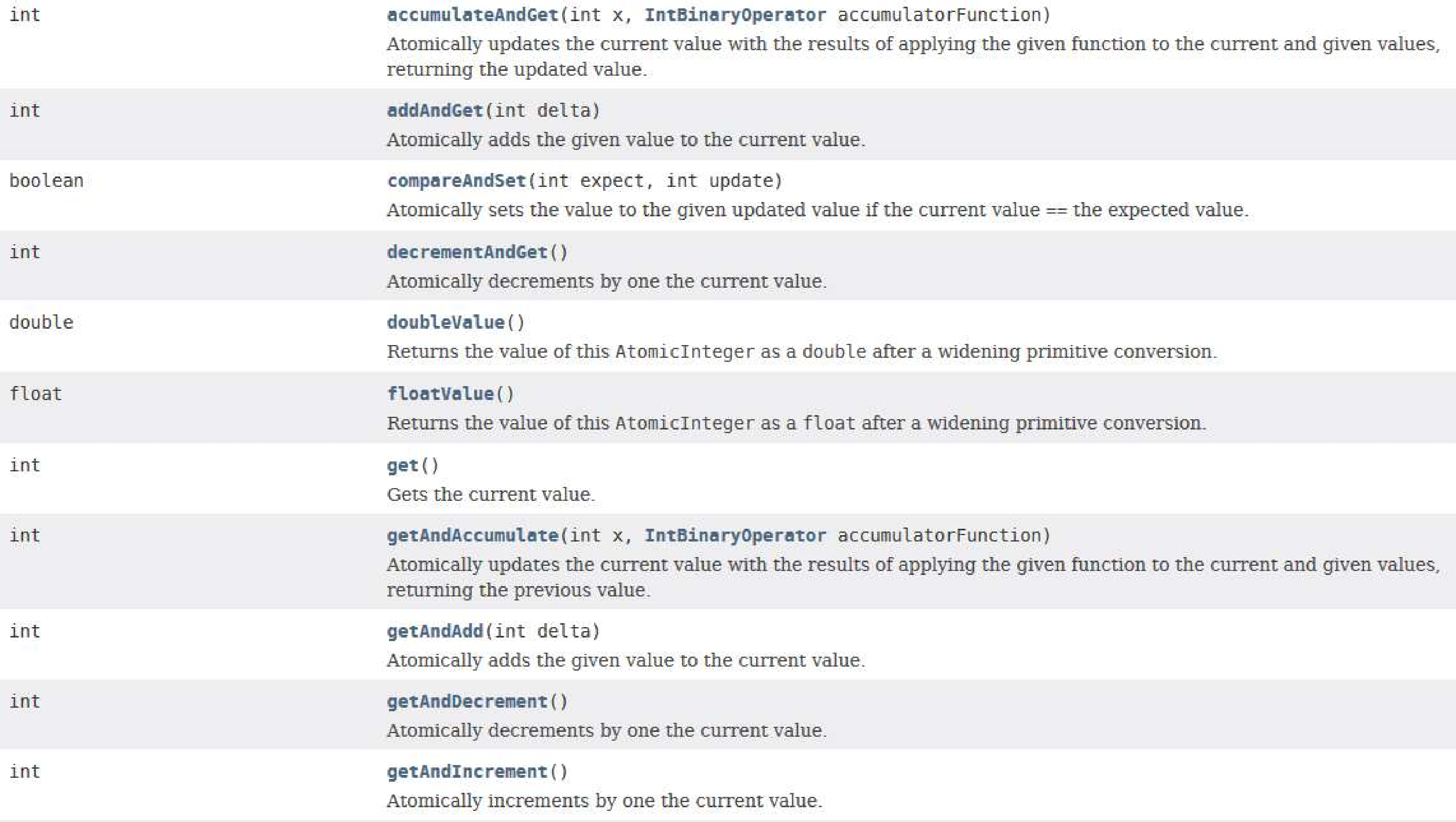
Assignments to variables of primitive types
except for long and double are atomic.

Remember: only assignments!
Statements like x += y or even x++ are NOT guaranteed to be atomic, even for the int type.

If you need atomicity in such a case,
use synchronization or a special atomic type from
the java.util.concurrent.atomic package.

java.util.concurrent.atomic.AtomicInteger

(extends Number implements Serializable)



int	accumulateAndGet(int x, IntBinaryOperator accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value.
int	addAndGet(int delta) Atomically adds the given value to the current value.
boolean	compareAndSet(int expect, int update) Atomically sets the value to the given updated value if the current value == the expected value.
int	decrementAndGet() Atomically decrements by one the current value.
double	doubleValue() Returns the value of this AtomicInteger as a double after a widening primitive conversion.
float	floatValue() Returns the value of this AtomicInteger as a float after a widening primitive conversion.
int	get() Gets the current value.
int	getAndAccumulate(int x, IntBinaryOperator accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the previous value.
int	getAndAdd(int delta) Atomically adds the given value to the current value.
int	getAndDecrement() Atomically decrements by one the current value.
int	getAndIncrement() Atomically increments by one the current value.

■ ■ ■

Guarded blocks

[http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html]

Suppose, for example, that *guardedJoy()* is a method that must not proceed until a shared variable *joy* has been set by another thread.

Do you like the following piece of code?

```
public void guardedJoy()
{
    while(!joy) { } // empty loop
     System.out.println("Joy has been achieved!");
}
```

This is bad. CPU cycles are wasted.

Guarded blocks, cont’d

[http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html]

```
// this is a correct way
public synchronized guardedJoy() {
    // This guard only loops once for each special event,
    // which may not be the event we're waiting for.
    while(!joy) {
        try { wait(); }
        catch (InterruptedException e) { }
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

We expect that some other thread will eventually invoke smth like this:
public synchronized notifyJoy() { joy = true; notifyAll(); }

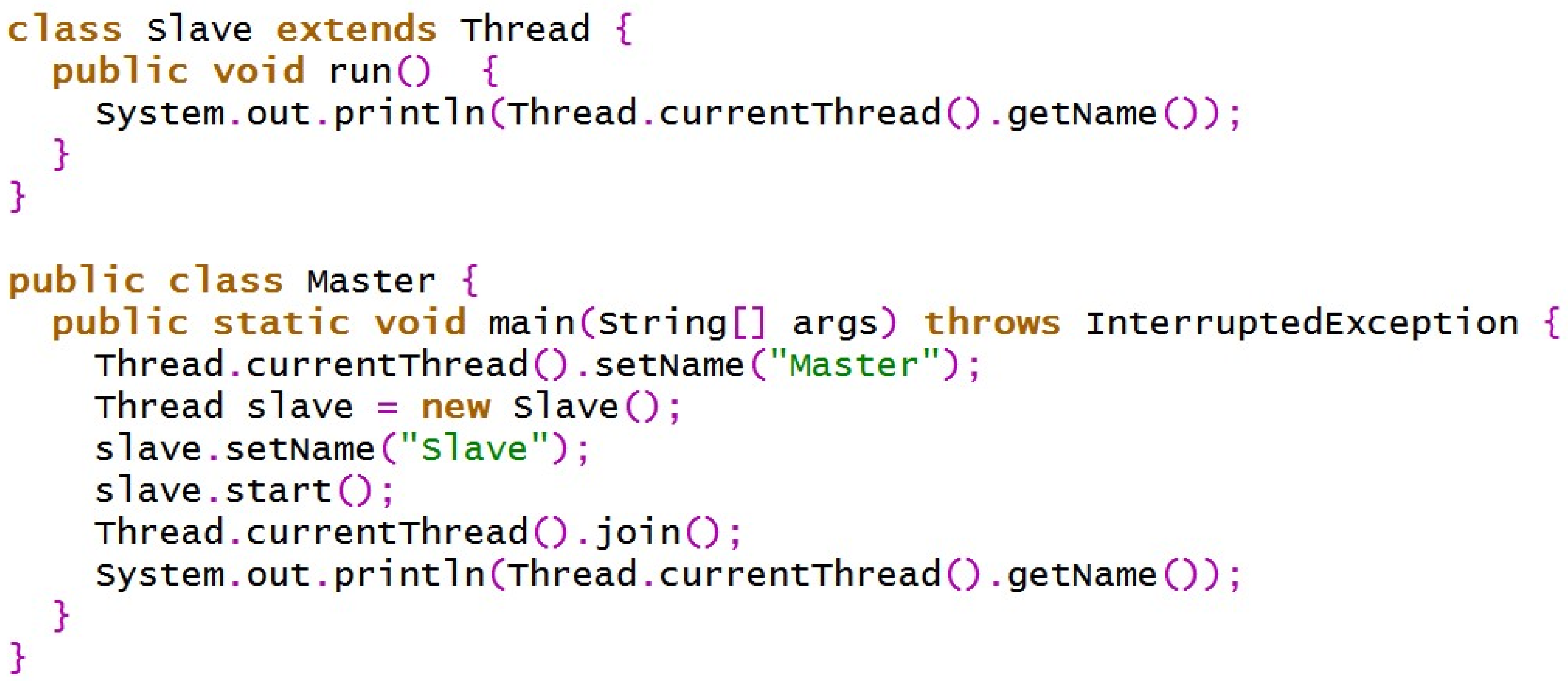
Note this kind of solution was used for the producer / consumer problem.

Thread.join()

Thread.join() method waits until the thread on which it is called terminates (dies).
I.e., t.join() causes the current thread to pause execution until t’s thread terminates.

Overloads of join allow the programmer to specify a waiting period.
However, as with sleep, join is dependent on the OS for timing,
so you should not assume that join will wait exactly
as long as you specify.

Join question.
What’s the result of the following program?

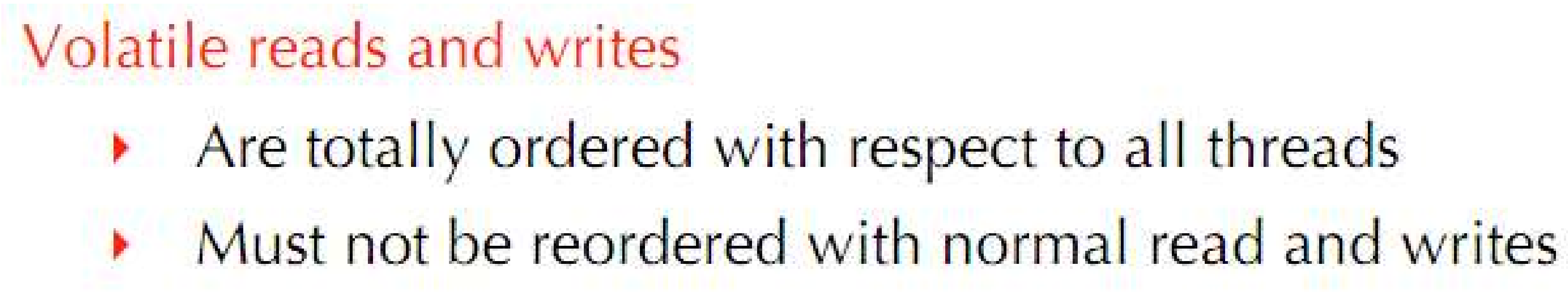


```
class Slave extends Thread {
  public void run()  {
    System.out.println(Thread.currentThread().getName());
  }
}

public class Master {
  public static void main(String[] args) throws InterruptedException {
    Thread.currentThread().setName("Master");
    Thread slave = new Slave();
    slave.setName("Slave");
    slave.start();
    Thread.currentThread().join();
    System.out.println(Thread.currentThread().getName());
  }
}
```

Answer: prints "Slave" and hangs (blocks infinitely on the Thread.currentThread().join() line).

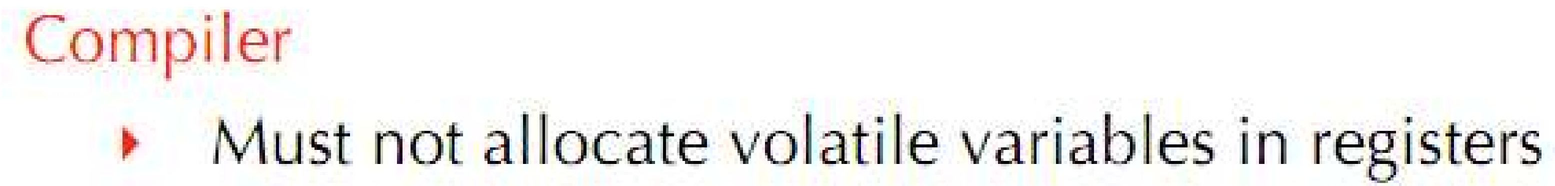
Volatile fields

Fields can be declared with a keyword volatile.



Volatile reads and writes

- Are totally ordered with respect to all threads
- Must not be reordered with normal read and writes



Compiler

- Must not allocate volatile variables in registers

Every read of a volatile variable will be read from main memory, and not from a register or the CPU cache,
and that every write to a volatile variable
will be written to main memory,
and not just to the CPU cache.

http://www.lst.inf.ethz.ch/teaching/lectures/hs12/2800/slides/2012_11_26Jmm.pdf

Since Java 5 the keyword *volatile* means more

If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable.
In other words: not only the volatile variable, but also all other variables changed by Thread A before writing to the volatile variable are flushed to main memory.

The reading and writing instructions of volatile variables cannot be reordered by the JVM (the JVM may reorder instructions for performance reasons as long as the JVM detects no change in program behavior from the reordering).
Instructions before and after can be reordered, but the volatile read or write cannot be mixed with these instructions.
Whatever instructions follow a read or write of a volatile variable are guaranteed to happen after the read or write.

http://tutorials.jenkov.com/java-concurrency/volatile.html

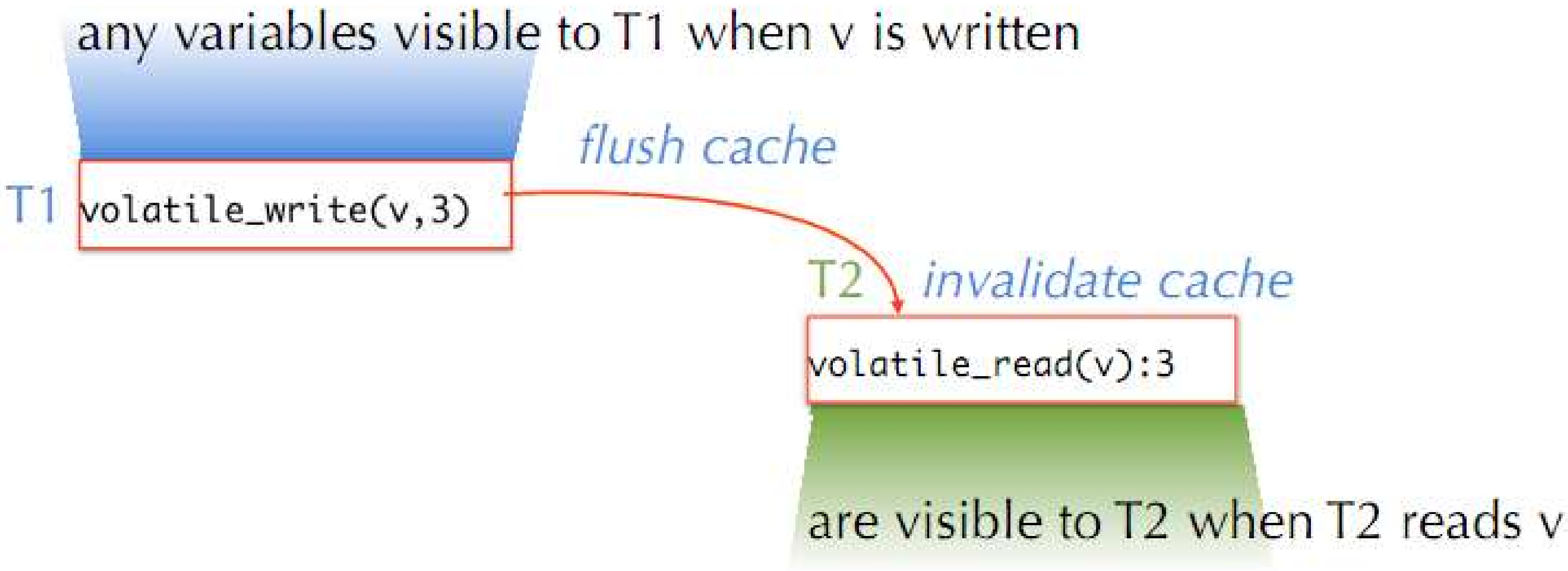
Java Memory Model (JMM) semantics

Volatile write

- Same semantics as unlock

Volatile read

- Same semantics as lock



any variables visible to T1 when v is written

flush cache

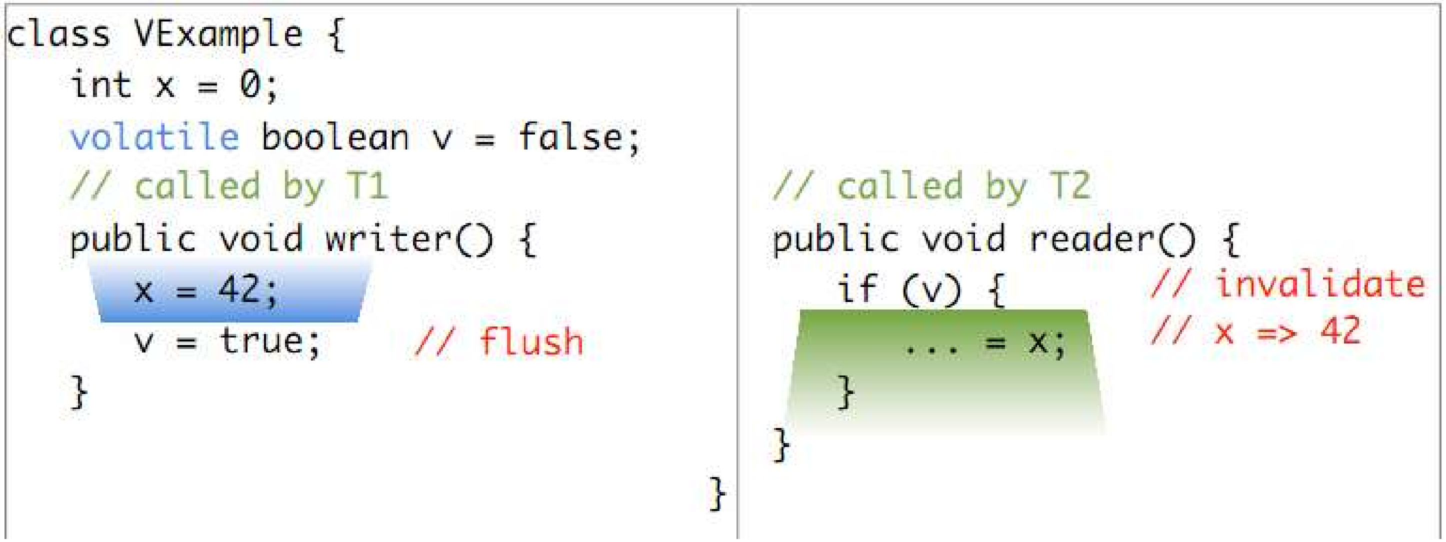
T1 `volatile_write(v,3)`

T2 invalidate cache

`volatile_read(v):3`

are visible to T2 when T2 reads v

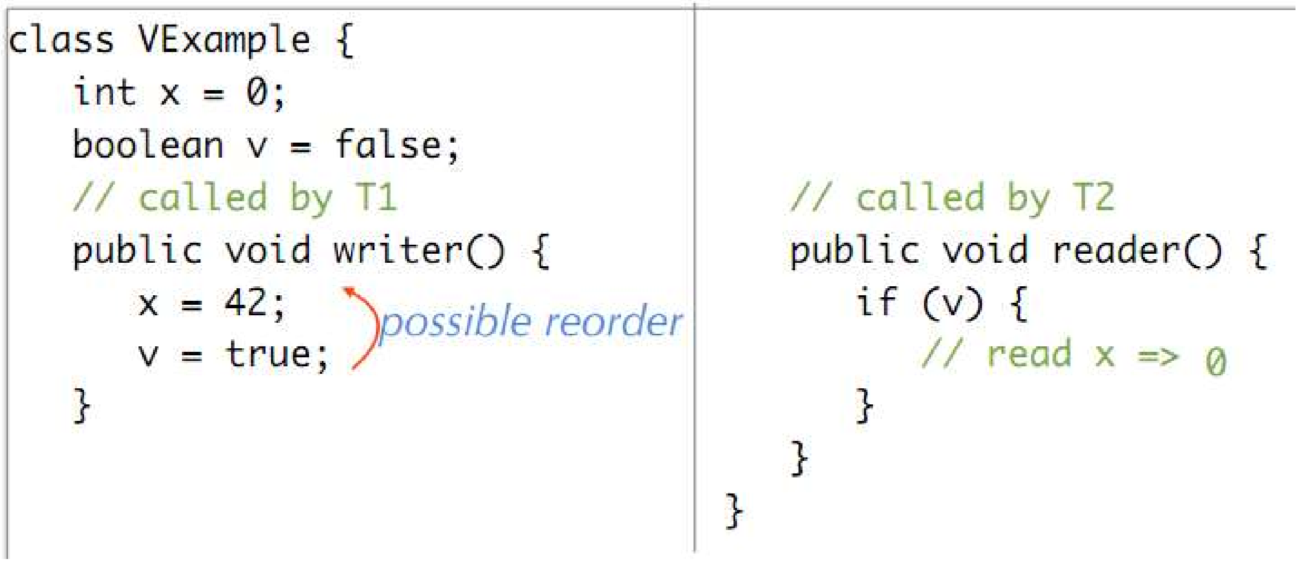
volatile example



```
class VExample {
   int x = 0;
   volatile boolean v = false;
   // called by T1
   public void writer() {
      x = 42;
      v = true;      // flush
   }
                                      }
```

```
// called by T2
public void reader() {
   if (v) {         // invalidate
                    // x => 42
      ... = x;
   }
}
```

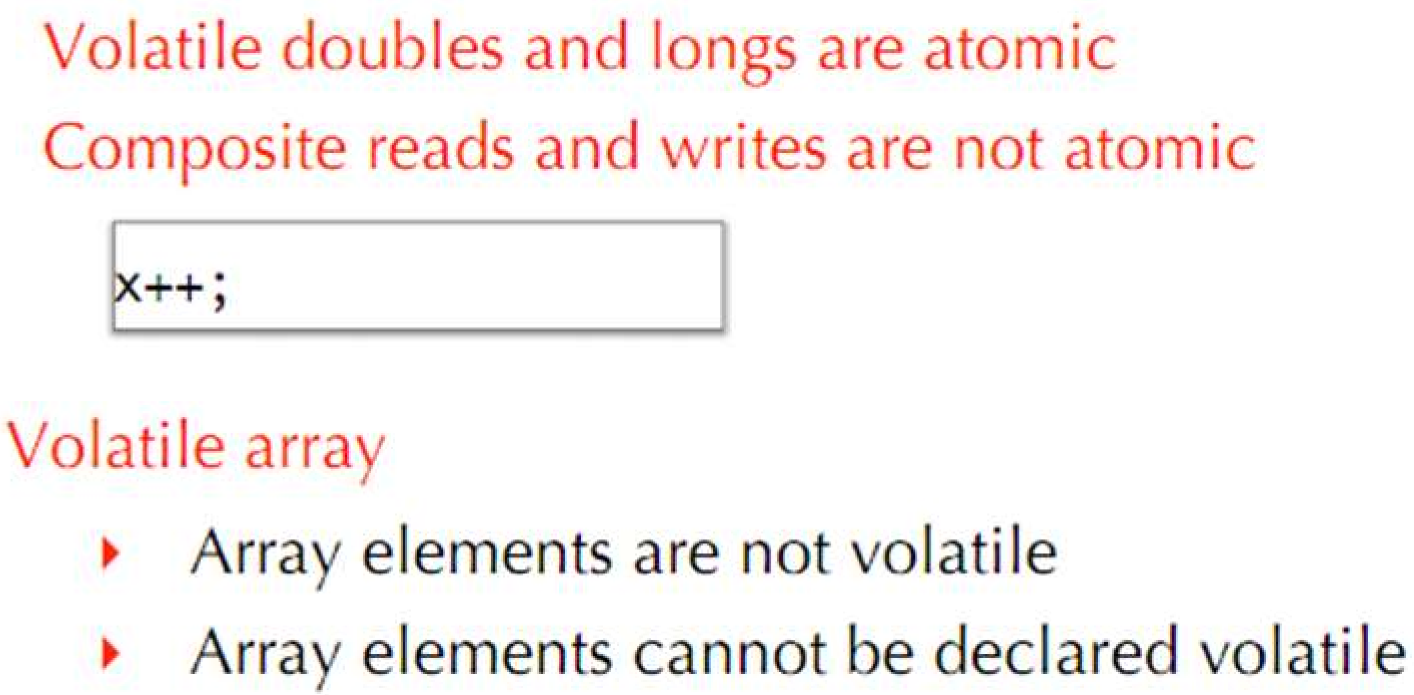
Below: without volatile



```
class VExample {
   int x = 0;
   boolean v = false;
   // called by T1
   public void writer() {
      x = 42;
      v = true;   possible reorder
   }
```

```
   // called by T2
   public void reader() {
      if (v) {
         // read x => 0
      }
   }
}
```

More on volatiles



Volatile doubles and longs are atomic

Composite reads and writes are not atomic

```
x++;
```

Volatile array

- Array elements are not volatile
- Array elements cannot be declared volatile

→ AtomicIntegerArray or AtomicLongArray, or AtomicReferenceArray

AtomicIntegerArray implements an int array whose slots can be accessed with volatile semantics, via get()/set().
Calling arr.set(x, y) from one thread will then guarantee that another thread calling arr.get(x) will read the value y.

StringBuffer, StringBuilder

Scenario: you want to modify a string.
You know already StringBuffer and StringBuilder.
No difference between them?

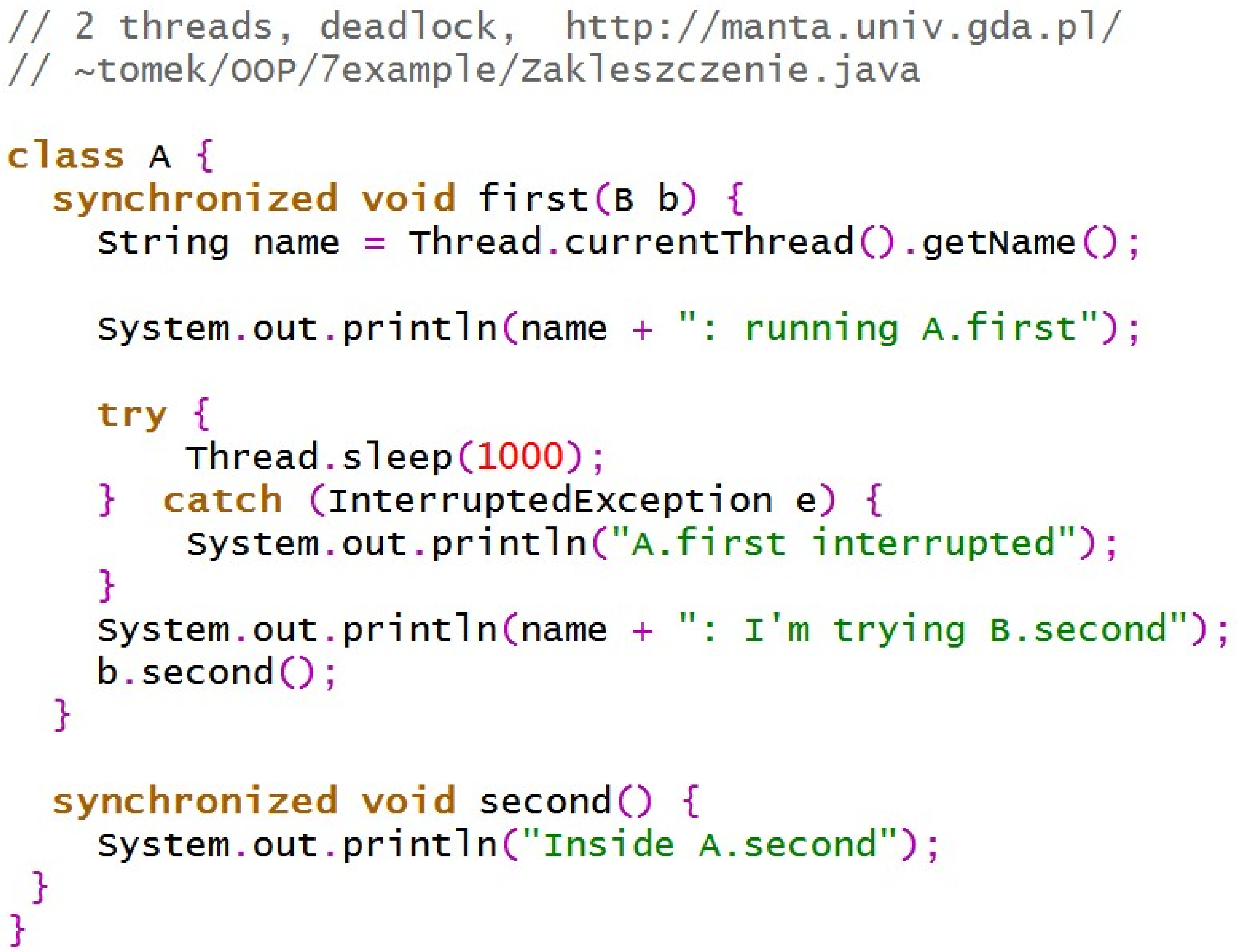
If the (mutable) string will be shared between threads,
then use the StringBuffer class:

- you’re sure that the string is updated correctly,
- ...but the methods may execute slower.

If the (mutable) string will NOT be shared between threads,
use the StringBuilder class:

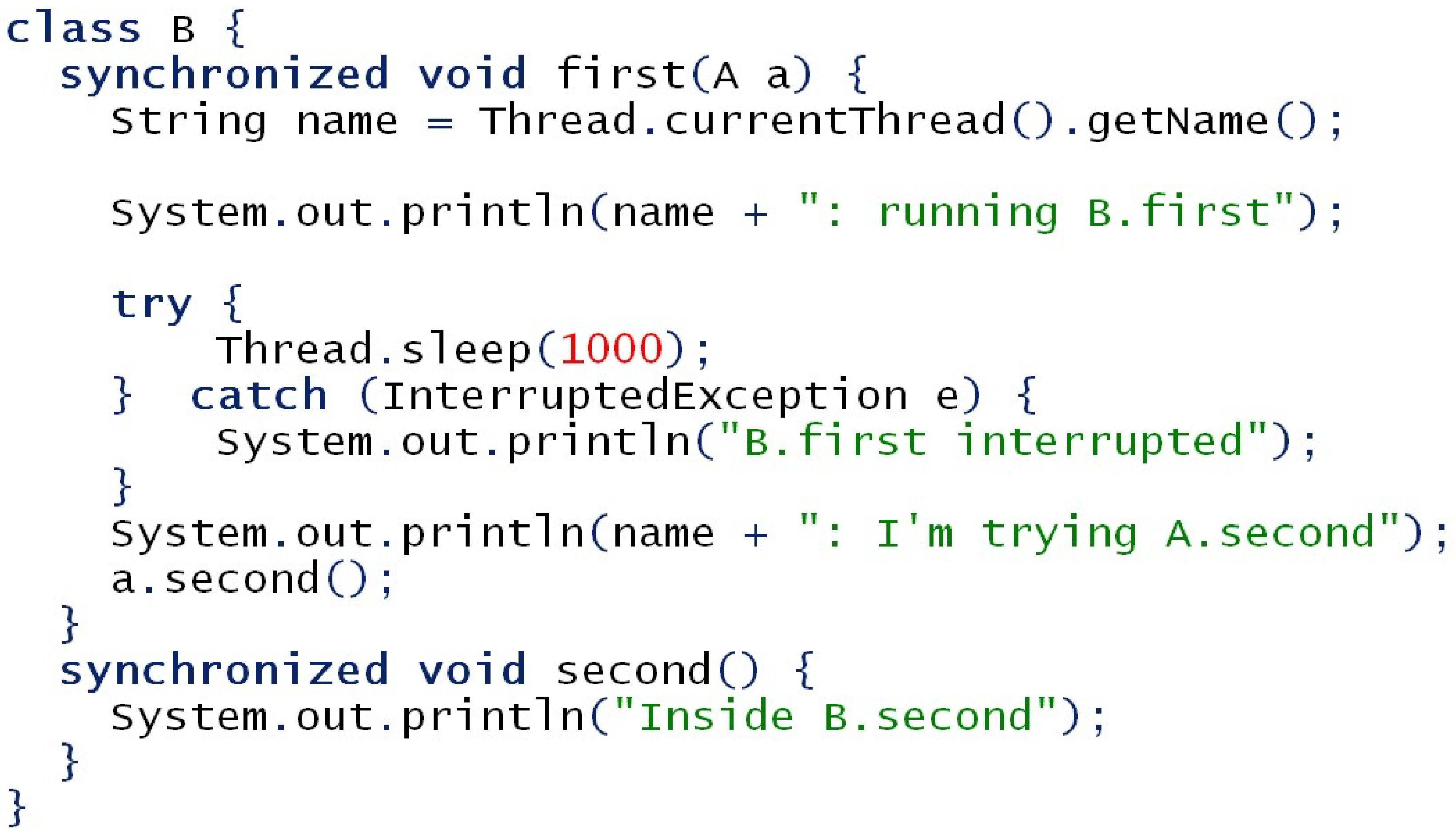
- no overhead of synchronization → faster.

Deadlock example (1/3)



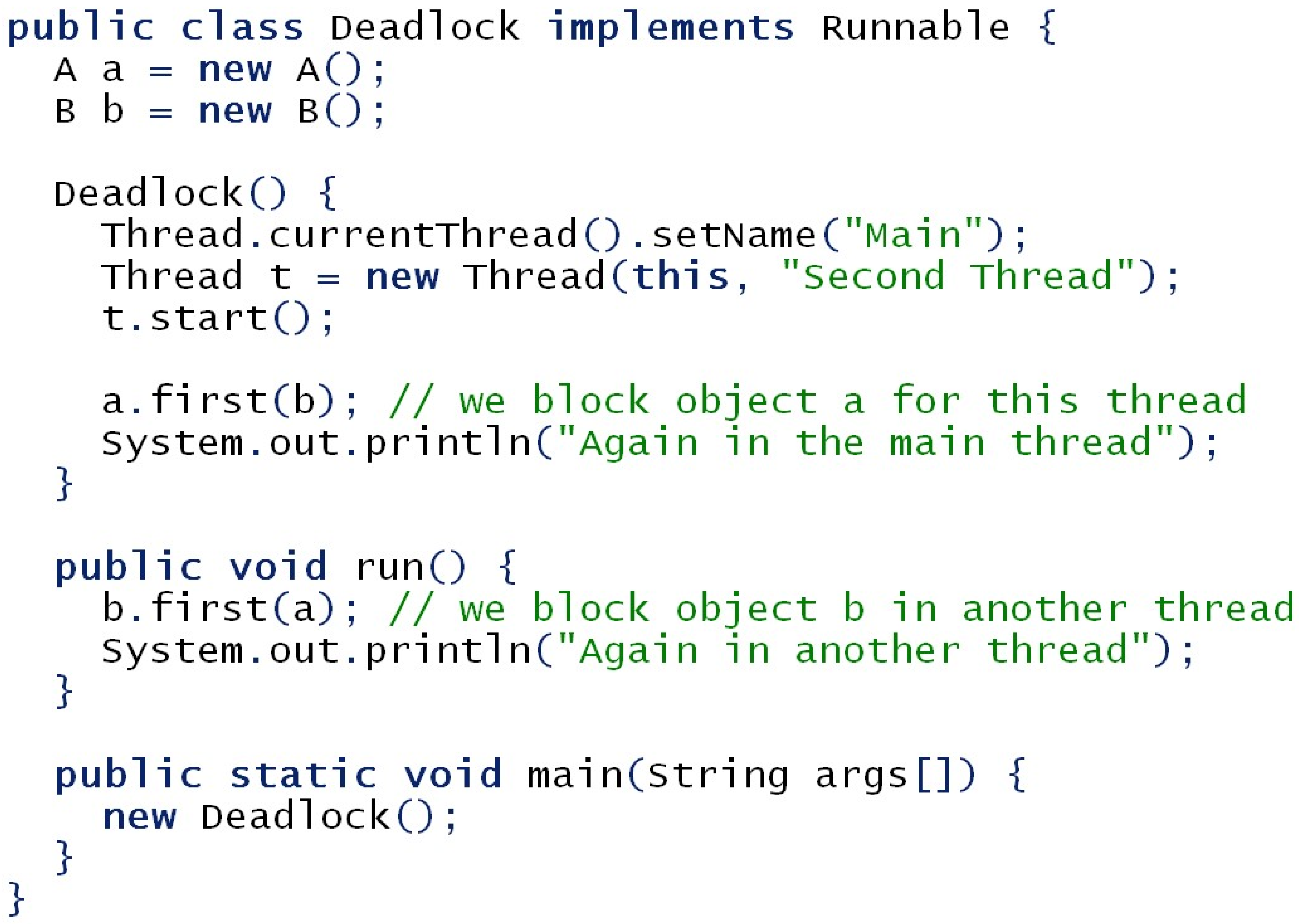
```
// 2 threads, deadlock,  http://manta.univ.gda.pl/
// ~tomek/OOP/7example/Zakleszczenie.java

class A {
  synchronized void first(B b) {
    String name = Thread.currentThread().getName();

    System.out.println(name + ": running A.first");

    try {
        Thread.sleep(1000);
    }  catch (InterruptedException e) {
        System.out.println("A.first interrupted");
    }
    System.out.println(name + ": I'm trying B.second");
    b.second();
  }

  synchronized void second() {
    System.out.println("Inside A.second");
 }
}
```

Deadlock example (2/3)



```
class B {
  synchronized void first(A a) {
    String name = Thread.currentThread().getName();

    System.out.println(name + ": running B.first");

    try {
        Thread.sleep(1000);
    }  catch (InterruptedException e) {
        System.out.println("B.first interrupted");
    }
    System.out.println(name + ": I'm trying A.second");
    a.second();
  }
  synchronized void second() {
    System.out.println("Inside B.second");
  }
}
```

Deadlock example (3/3)



```java
public class Deadlock implements Runnable {
  A a = new A();
  B b = new B();

  Deadlock() {
    Thread.currentThread().setName("Main");
    Thread t = new Thread(this, "Second Thread");
    t.start();

    a.first(b); // we block object a for this thread
    System.out.println("Again in the main thread");
  }

  public void run() {
    b.first(a); // we block object b in another thread
    System.out.println("Again in another thread");
  }

  public static void main(String args[]) {
    new Deadlock();
  }
}
```

java.util.concurrent (since Java 1.5)

It is hard to write multi-threaded apps…
…especially if the number of threads is large
(SCALED concurrency!).

The package java.util.concurrent
introduced many useful classes, for

- separation of threads and tasks,
- thread pools handled by executors,
- effective locks,
- blocking queues,
- concurrent collections,
- atomic types,

and more.

Separate tasks from threads

A task is something to do.

A thread is a unit of processing (smth physical, not conceptual), a technical detail…

```
public interface Executor {
        void execute(Runnable r);
}
```

Implementations of interface Executor should create and run threads. They will follow some general strategy (e.g., max number of threads in a pool can be specified).

Executors class

We don't have to create our own Executor classes (i.e. classes implementing the interface Executor). Instead, we can made use of ready-made executors, produced by static factory methods from Executors (they create objects implementing ExecutorService).

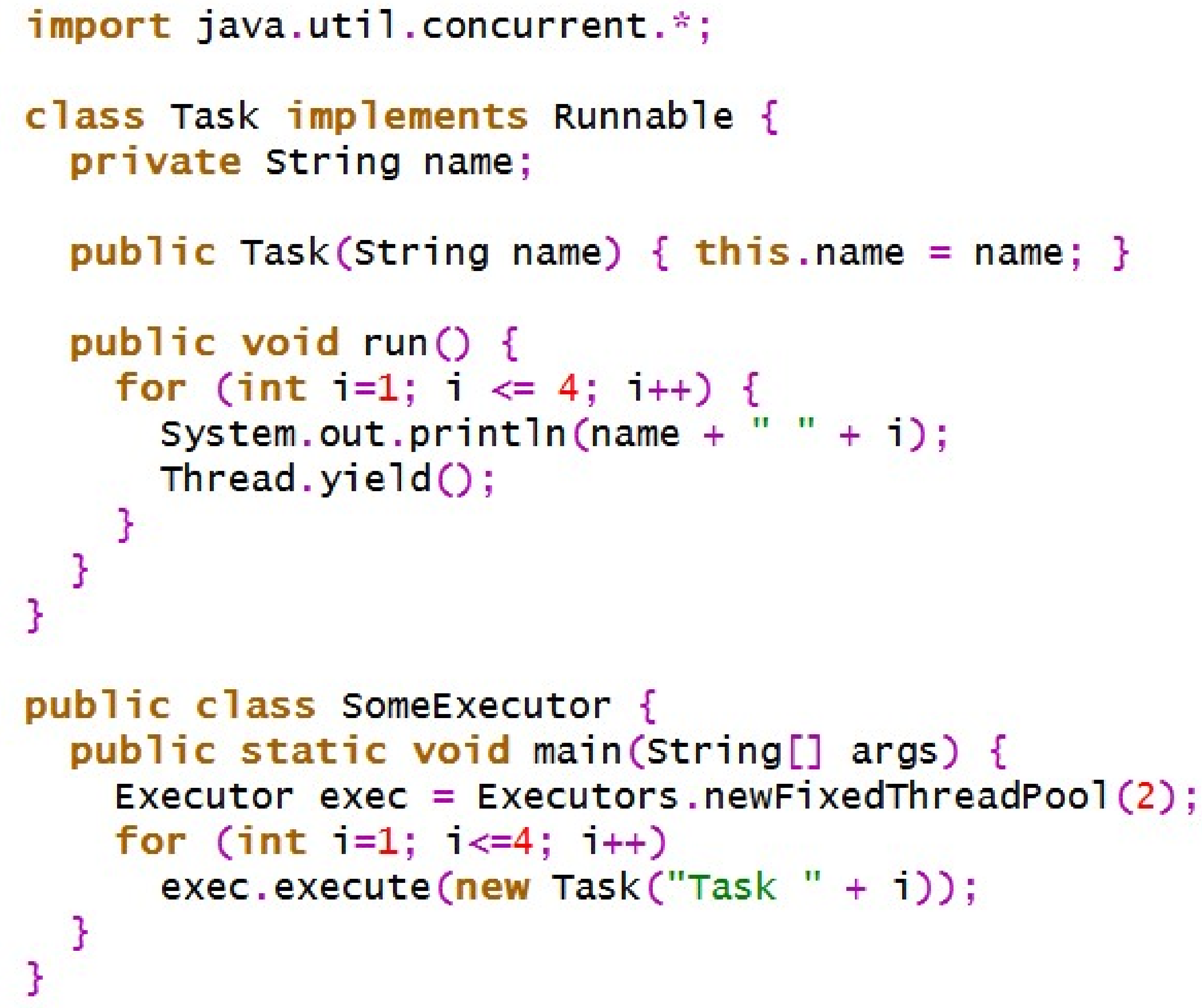
Executors.newSingleThreadExecutor()
runs the given tasks one by one (one thread at a time)

Executors.newFixedThreadPool(…)
given thread pool size (if more tasks, they'll wait in a queue)

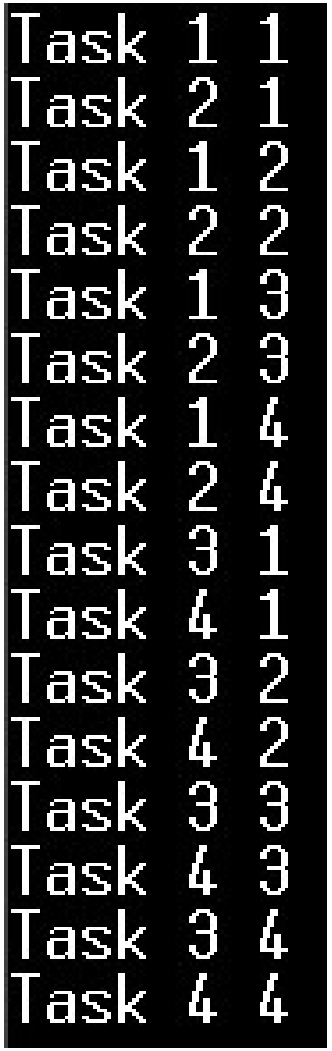
Executors.newCachedThreadPool()
dynamic (=variable-size) thread pool;
will reuse previously constructed threads when available

Executors in action, example

[http://edu.pjwstk.edu.pl/wyklady/zap/scb/W8/W8.htm]



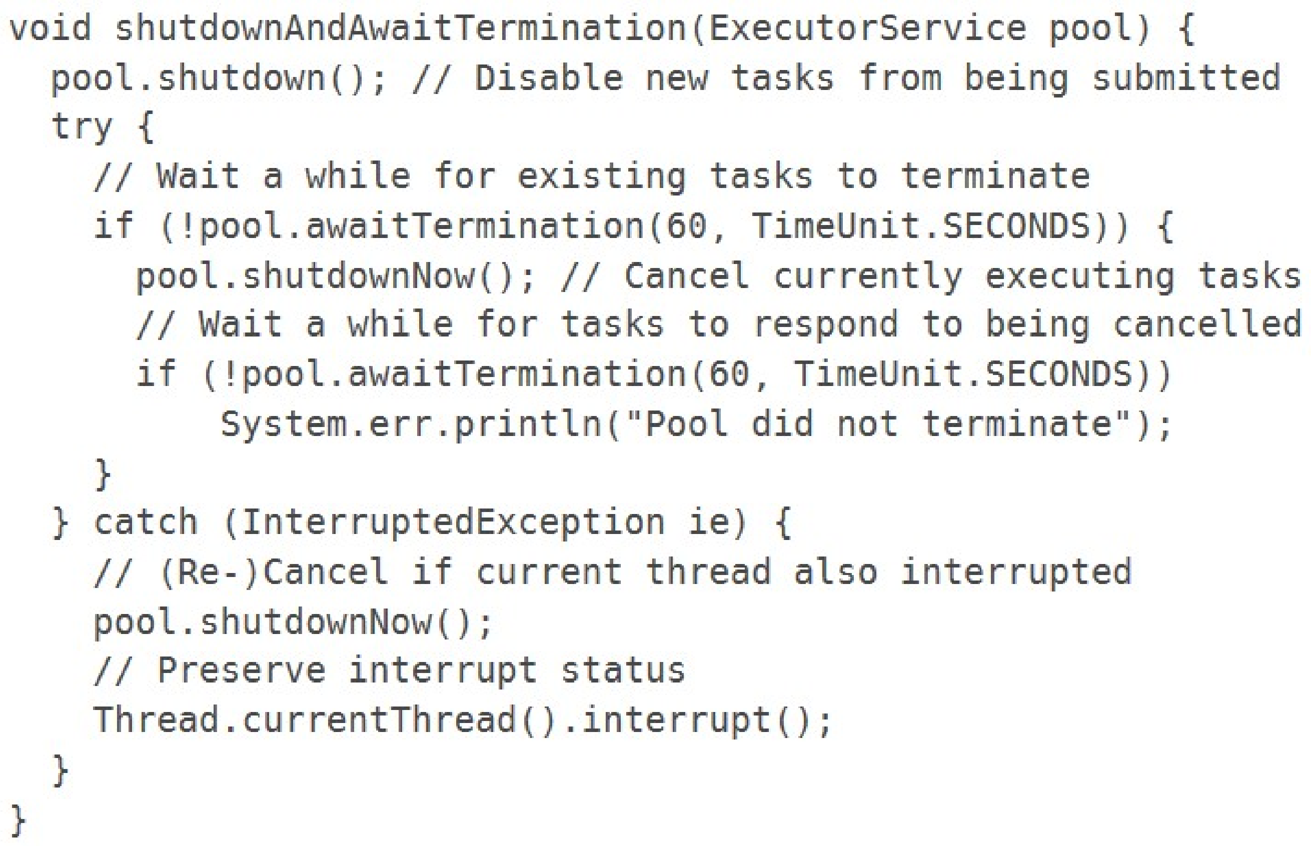
```java
import java.util.concurrent.*;

class Task implements Runnable {
  private String name;

  public Task(String name) { this.name = name; }

  public void run() {
    for (int i=1; i <= 4; i++) {
      System.out.println(name + " " + i);
      Thread.yield();
    }
  }
}

public class SomeExecutor {
  public static void main(String[] args) {
    Executor exec = Executors.newFixedThreadPool(2);
    for (int i=1; i<=4; i++)
      exec.execute(new Task("Task " + i));
  }
}
```



```
Task 1 1
Task 2 1
Task 1 2
Task 2 2
Task 1 3
Task 2 3
Task 1 4
Task 2 4
Task 3 1
Task 4 1
Task 3 2
Task 4 2
Task 3 3
Task 4 3
Task 3 4
Task 4 4
```

ExecutorService

shutdown() method (in ExecutorService interface)
– the object will reject new tasks (but won’t close the currently run ones)



```
void shutdownAndAwaitTermination(ExecutorService pool) {
  pool.shutdown(); // Disable new tasks from being submitted
  try {
    // Wait a while for existing tasks to terminate
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
      pool.shutdownNow(); // Cancel currently executing tasks
      // Wait a while for tasks to respond to being cancelled
      if (!pool.awaitTermination(60, TimeUnit.SECONDS))
          System.err.println("Pool did not terminate");
    }
  } catch (InterruptedException ie) {
    // (Re-)Cancel if current thread also interrupted
    pool.shutdownNow();
    // Preserve interrupt status
    Thread.currentThread().interrupt();
  }
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

Callable tasks

Sometimes we wish to deal with tasks returning values (or possibly throwing exceptions, when needed).

```
public interface Callable<V> {
        V call() throws Exception
}
```

Something like Runnable, but with two major differences:

- returns a value,
- may throw a (checked) exception.

Future<V>

[http://download.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html]

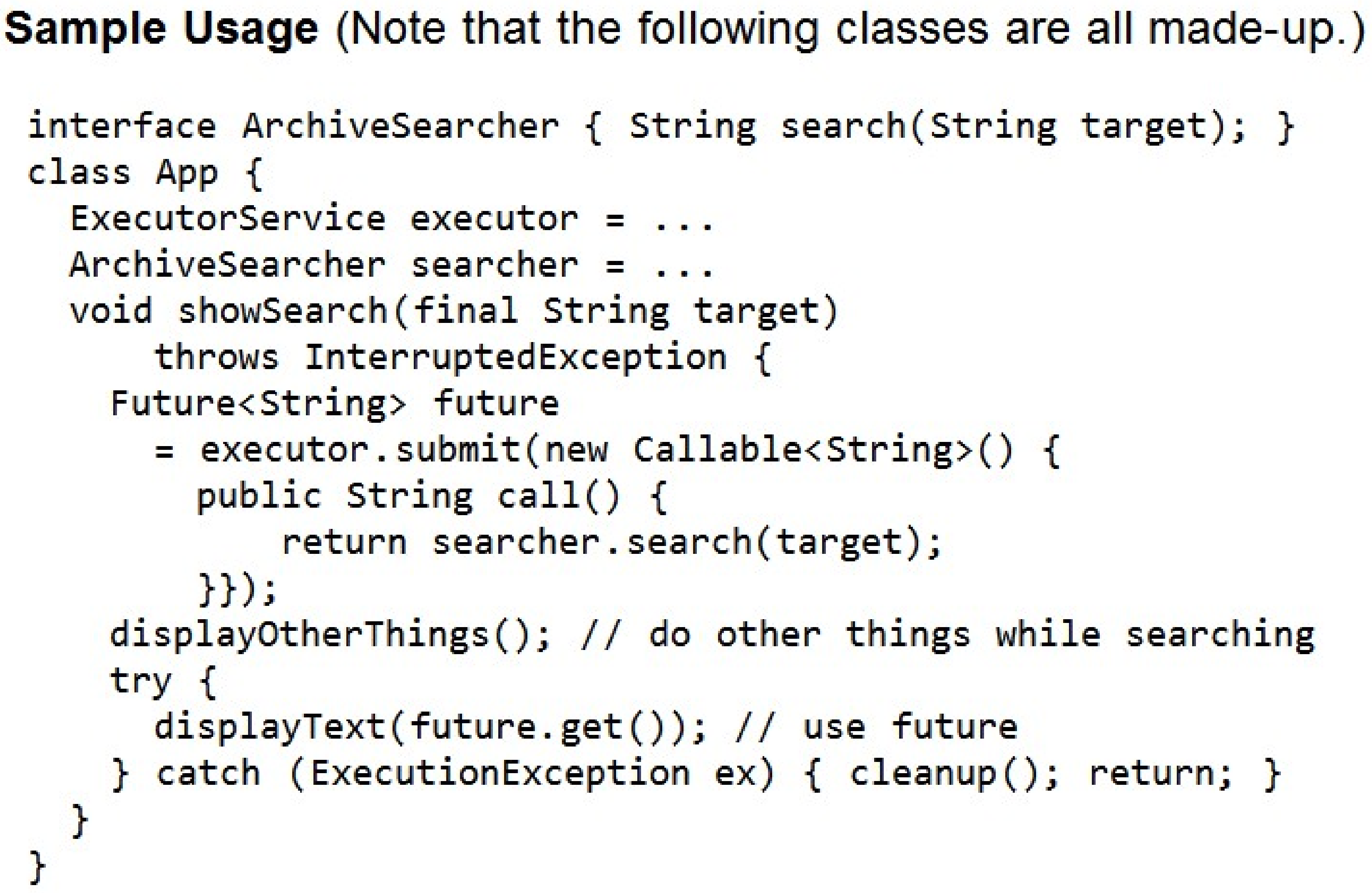
A Future represents the result of an asynchronous call (computation).

There are methods to check
if the computation is complete (isDone()),
to wait until it is complete (get(long timeout, TimeUnit unit)),
to get its result (get()).

Method *cancel* is also provided (won’t work once the computation was completed).

Future example

[http://download.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html]



Sample Usage (Note that the following classes are all made-up.)

```
interface ArchiveSearcher { String search(String target); }
class App {
  ExecutorService executor = ...
  ArchiveSearcher searcher = ...
  void showSearch(final String target)
      throws InterruptedException {
    Future<String> future
      = executor.submit(new Callable<String>() {
        public String call() {
            return searcher.search(target);
        }});
    displayOtherThings(); // do other things while searching
    try {
      displayText(future.get()); // use future
    } catch (ExecutionException ex) { cleanup(); return; }
  }
}
```

Problem with Future (solution from Java 8)

[http://www.deadcoderising.com/java8-writing-asynchronous-code-with-completablefuture/]

The method get(…) in Future is blocking.
This doesn’t fit nicely asynchronous computation.

```
public class
CompletableFuture<T>
extends Object
implements Future<T>,
CompletionStage<T>
```

Simplest example:

```
CompletableFuture.supplyAsync(this::sendMsg);
```

supplyAsync takes a Supplier containing the code we want to execute asynchronously — in our case the sendMsg method.

CompletableFuture, cont’d

Now let’s add a callback where we notify about how the sending of the message went:

```
CompletableFuture.supplyAsync(this::sendMsg)
                    .thenAccept(this::notify);
```

In this way, we say what should happen when an async computation is done without waiting around for the result.

thenAccept is one of many ways to add a callback. It takes a Consumer (e.g., notify) which handles the result of the preceding comput. when done.

```
CompletableFuture.supplyAsync(this::failingMsg)
              .exceptionally(ex -> new Result(Status.FAILED))
              .thenAccept(this::notify);
```

exceptionally gives a chance to recover by taking an alternative function to be called if preceding calculation fails with an exception.

Task features

[http://edu.pjwstk.edu.pl/wyklady/zap/scb/W8/W8.htm]

• must allow for async execution in a separate thread
– i.e., must be Runnable,

• must be separable from its thread (separation of concerns)
– executors handle that,

• its results must be available (return value,
info if / what exception was thrown,
info if the task was cancelled during execution
or removed from the queue before its execution start)
– i.e. must be Future, its code must be Callable,

• there must be a way to cancel / break it
(not ‘touching’ a thread directly)
– i.e. must be Future.

FutureTask

[http://edu.pjwstk.edu.pl/wyklady/zap/scb/W8/W8.htm]

So, a task is an object implementing the interfaces Runnable and Future, and its code is in the method call() of a class implementing the interface Callable.

FutureTask class is provided, to wrap a Callable or Runnable object. It has methods get(), get(…), cancel(…), done(…), set(…) and more.

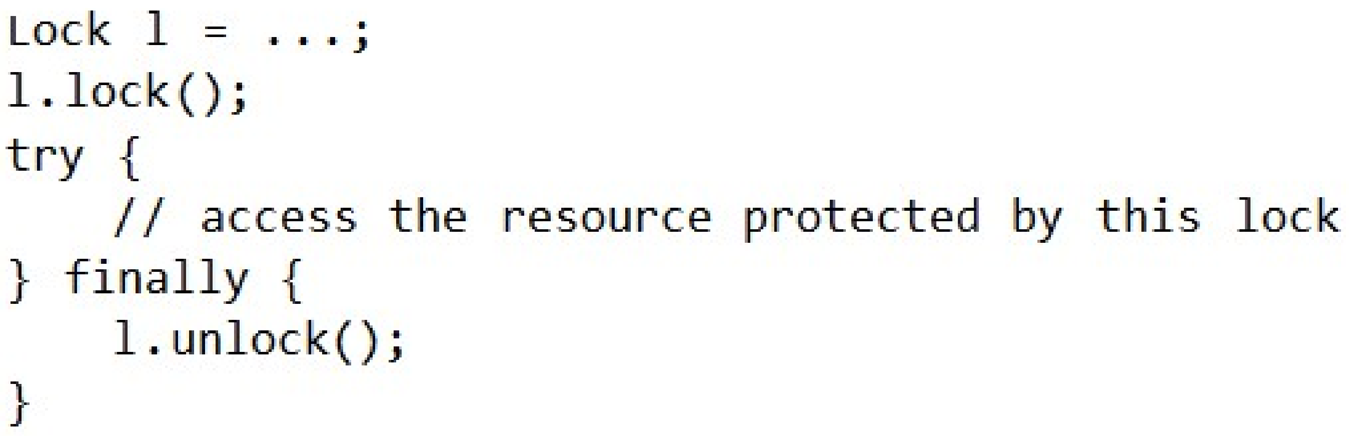
FutureTask implements Runnable. Because of that it can be submitted to an Executor for execution.

Lock interface

[http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/Lock.html]

3 classes in java.util.concurrent.locks to implement Lock interface: ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock.

Common idiom:



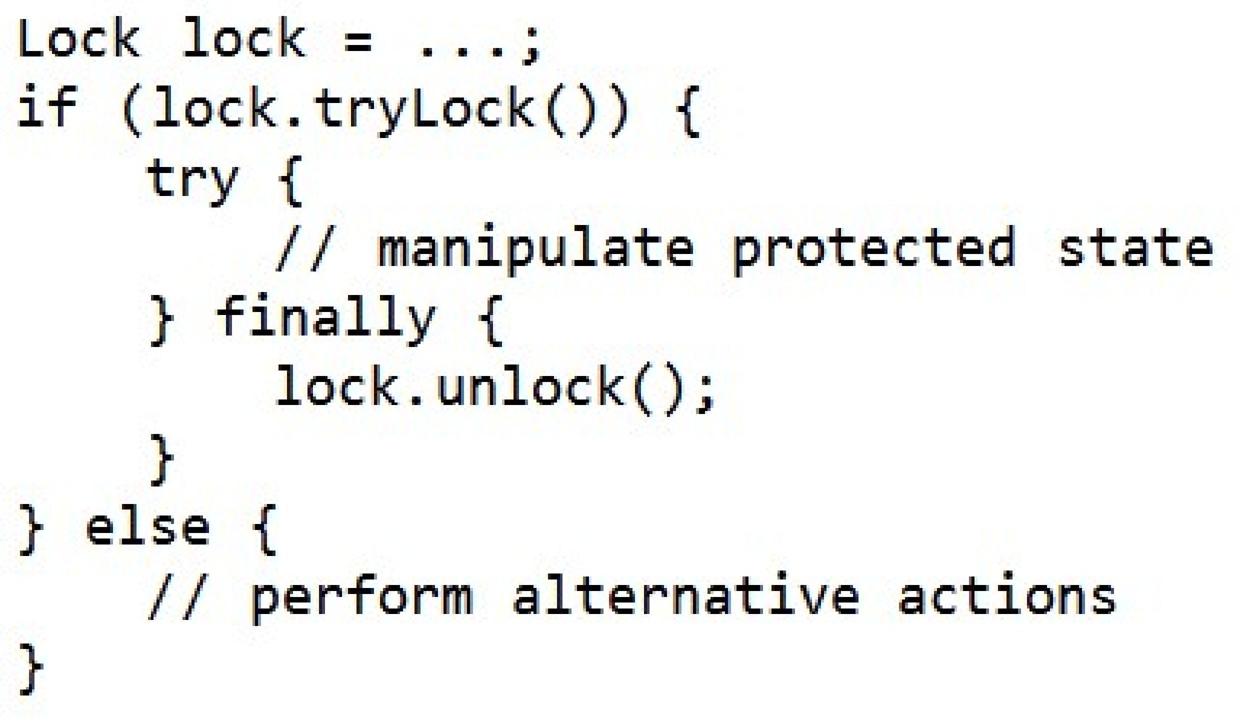
```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Why (a class implementing) Lock is better than *synchronized*?

- more readable code,
- (possibly) more effective in highly concurrent cases,
- may be passed to methods / constructors,
- may be locked / unlocked in different methods
(but by the same thread) – impossible with *synchronized*;
- tryLock() method – if another thread is in the critical region,
our thread may do smth else instead of blocking;
- tryLock(long, TimeUnit) – to attempt the lock with timeout.

tryLock() example

[https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html]



```
Lock lock = ...;
if (lock.tryLock()) {
     try {
          // manipulate protected state
     } finally {
          lock.unlock();
     }
} else {
     // perform alternative actions
}
```

boolean tryLock():
acquires the lock only if it is free at the time of invocation
(and returns true)

CyclicBarrier

CyclicBarrier class is a barrier (=synchronization mechanism) that all threads must wait at, until n threads reach it, before any of the threads can continue.

Create a cyclic barrier:
CyclicBarrier barrier = new CyclicBarrier(3);
or
CyclicBarrier barrier = new CyclicBarrier(3, r);
// r is a Runnable (action) executed after
// the last (of 3) thread arrives

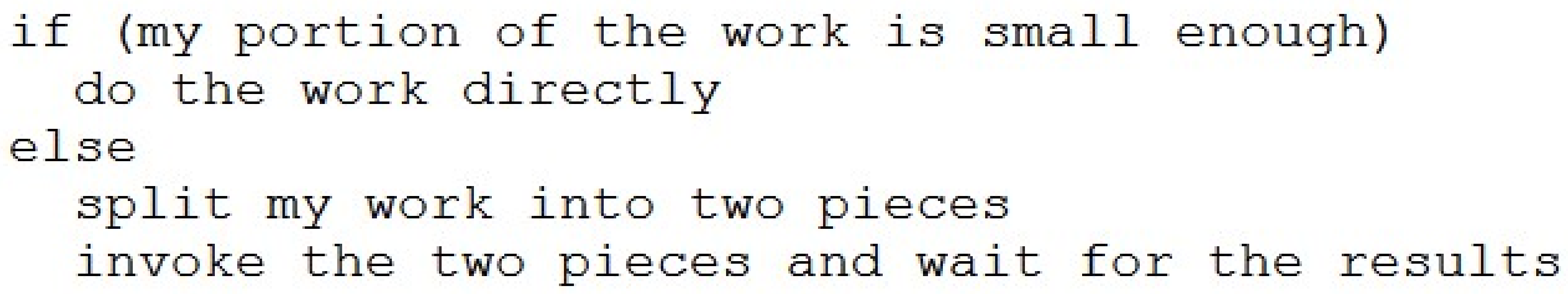
How a thread waits at a CyclicBarrier:
barrier.await();
or: barrier.await(5, TimeUnit.SECONDS);

See example:
http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html

Fork/Join framework

The fork/join framework is an implementation
of the ExecutorService interface
that helps us take advantage of multiple processors.
It is designed for work that can be
broken into smaller pieces recursively.

Work-stealing idea: worker threads that run out of things to do can steal tasks from other threads that are still busy.



```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

4 main Fork/Join classes (from java.util.concurrent)

ForkJoinPool: we create exactly one of these to run all your fork-join tasks in the whole program

RecursiveTask<V>: we run a subclass of this in a pool and have it return a result

RecursiveAction: just like RecursiveTask except it does not return a result

ForkJoinTask<V>: superclass of
RecursiveTask<V> and RecursiveAction.
fork and join are methods defined in this class.
We won't use this class directly, but it is the class with most of the useful javadoc documentation

http://homes.cs.washington.edu/~djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html

Basic syntax

RecursiveTask<V> has an abstract method
V compute(), which is often not called directly
(in a concrete class extending RecursiveTask<V>),
but is called by method fork().

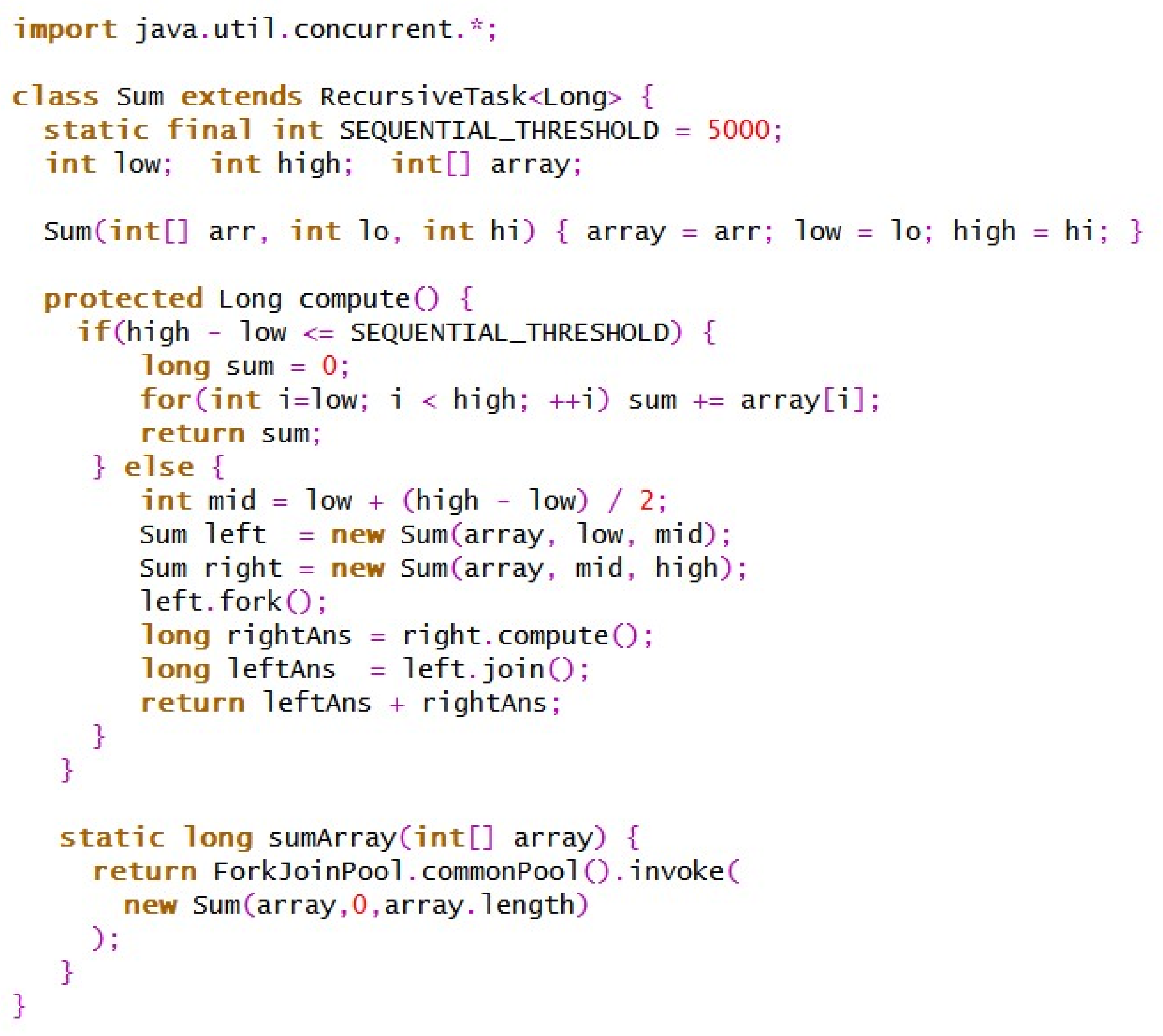
After our ForkJoinTask subclass
(i.e., a subclass of RecursiveTask<V> or RecursiveAction)
is ready, we create the object that represents
all the work to be done and pass it to the invoke() method
of a ForkJoinPool instance.

Basic syntax, cont’d

The idea is that our compute method can create other RecursiveTask objects and have the pool run them in parallel.
First we create another object. Then we call its fork method.
That actually starts parallel computation – fork itself returns quickly, but more computation is now going on.
When you need the answer, you call the join method on the object you called fork on.
The join method will get you the answer from compute() that was figured out by fork.
If it is not ready yet, then join will block (i.e., not return) until it is ready.
So the point is to call fork “early” and call join “late”, doing other useful work in-between.

http://homes.cs.washington.edu/~djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html

Simple example

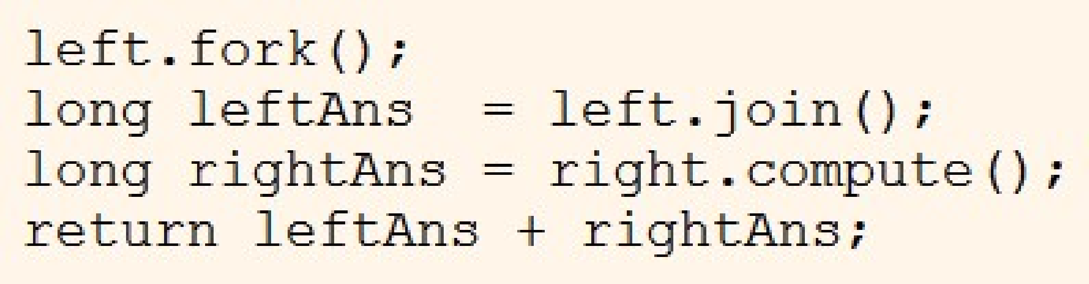


```
import java.util.concurrent.*;

class Sum extends RecursiveTask<Long> {
  static final int SEQUENTIAL_THRESHOLD = 5000;
  int low;  int high;  int[] array;

  Sum(int[] arr, int lo, int hi) { array = arr; low = lo; high = hi; }

  protected Long compute() {
    if(high - low <= SEQUENTIAL_THRESHOLD) {
        long sum = 0;
        for(int i=low; i < high; ++i) sum += array[i];
        return sum;
     } else {
        int mid = low + (high - low) / 2;
        Sum left  = new Sum(array, low, mid);
        Sum right = new Sum(array, mid, high);
        left.fork();
        long rightAns = right.compute();
        long leftAns  = left.join();
        return leftAns + rightAns;
     }
   }

   static long sumArray(int[] array) {
     return ForkJoinPool.commonPool().invoke(
       new Sum(array,0,array.length)
     );
   }
}
```

http://homes.cs.washington.edu/~djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html

Remember that calling join blocks until the answer is ready!

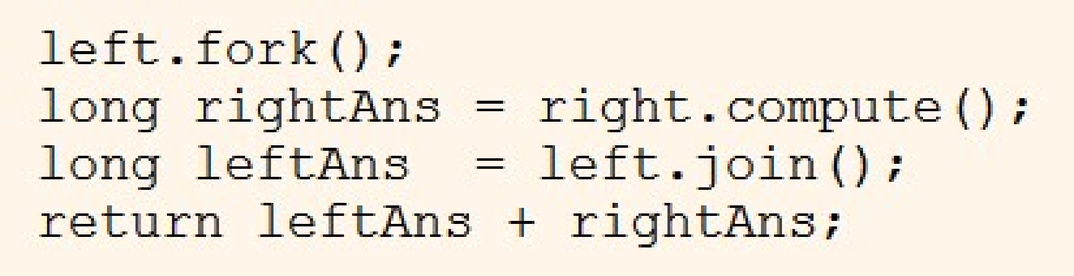
So, which of the following 3 code orders is correct?



```
left.fork();
long leftAns  = left.join();
long rightAns = right.compute();
return leftAns + rightAns;
```

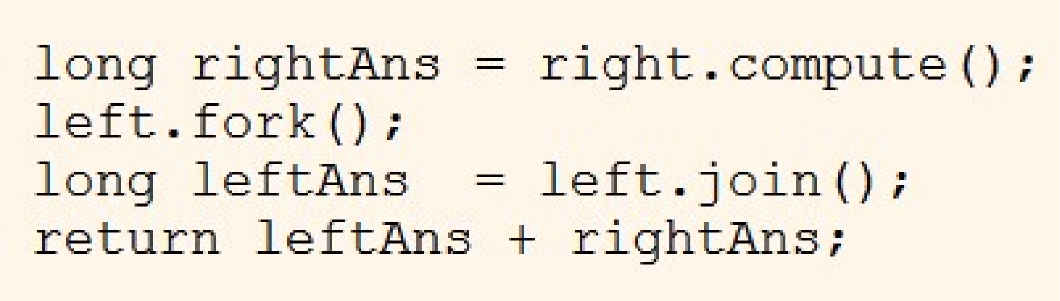
no parallelism!

since each step would completely compute the left before starting to compute the right



```
left.fork();
long rightAns = right.compute();
long leftAns  = left.join();
return leftAns + rightAns;
```

correct!

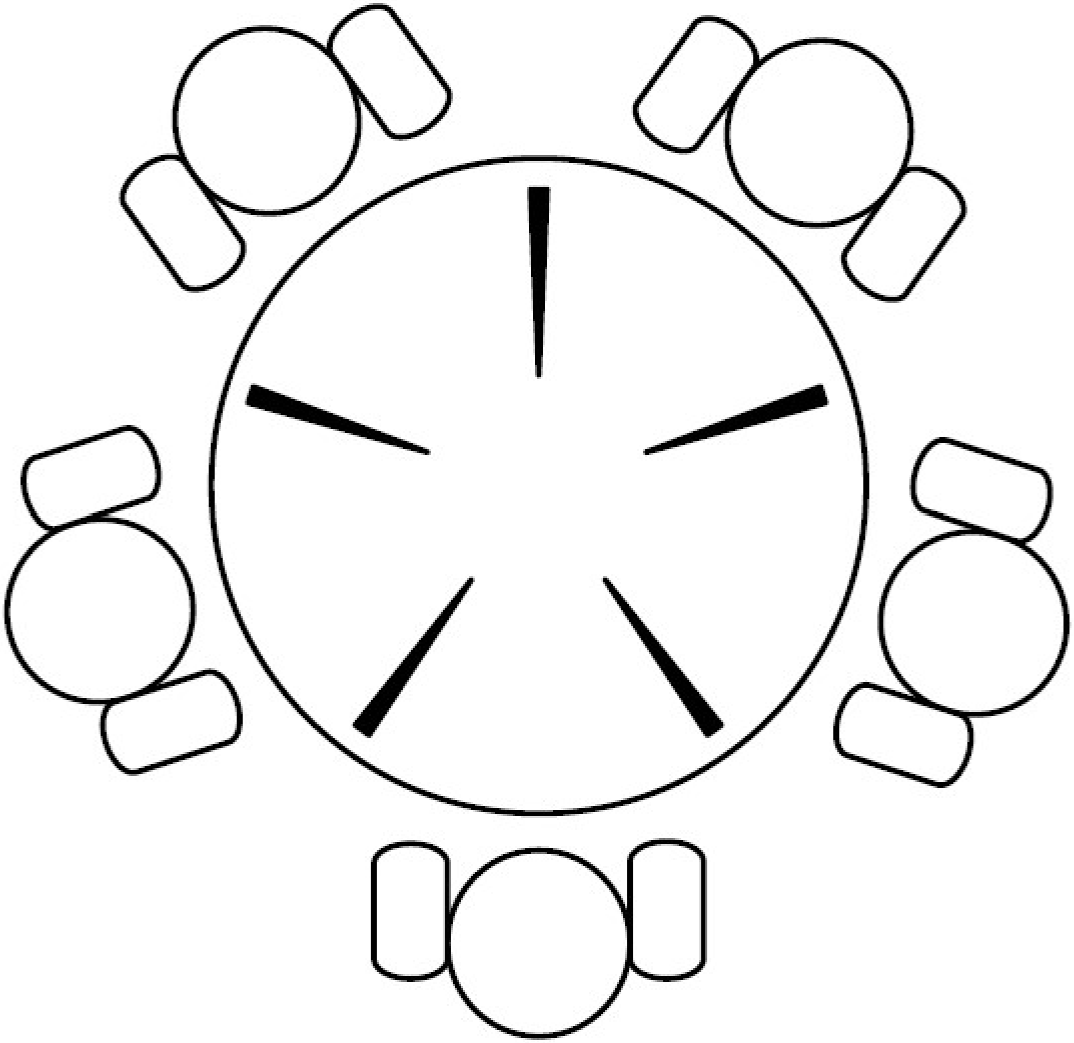


```
long rightAns = right.compute();
left.fork();
long leftAns  = left.join();
return leftAns + rightAns;
```

no parallelism!

since each step would completely compute the right before starting to compute the left

Dining philosophers



Thread priorities

A thread priority is an int from
Thread.MIN_PRIORITY (currently 1)
to Thread.MAX_PRIORITY (currently 10).

Higher priority – accessing the CPU more often.

Methods from class Thread():

```
void setPriority(int pr);
int getPriority();
```