Java Concurrency (Part 1)

Multi-Threaded Object-Oriented Programming

Dr Lee A. Christie

 @javaxnerd

 leechristie.com

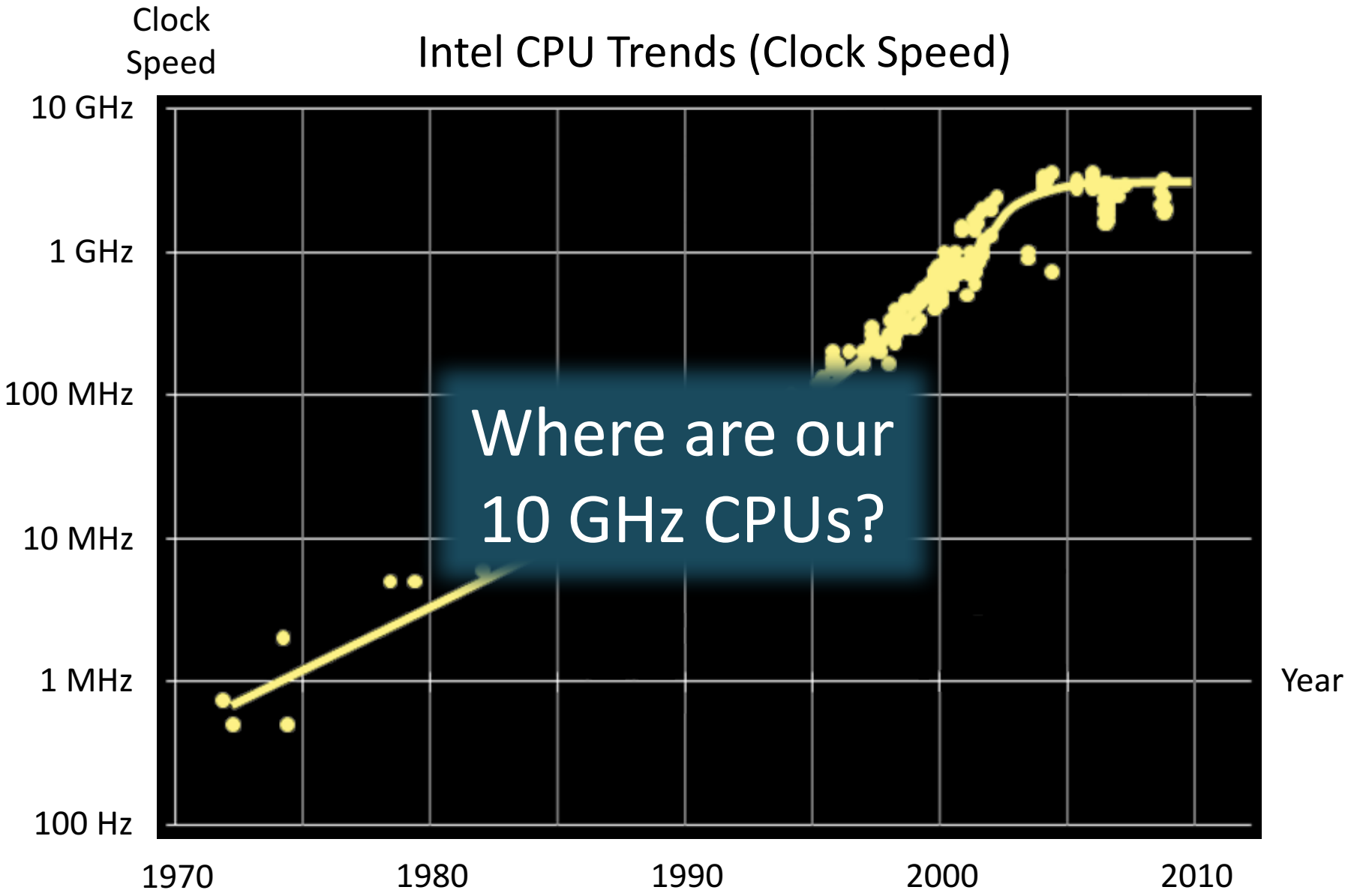
 lee@leechristie.com

Motivation

Moore's Law

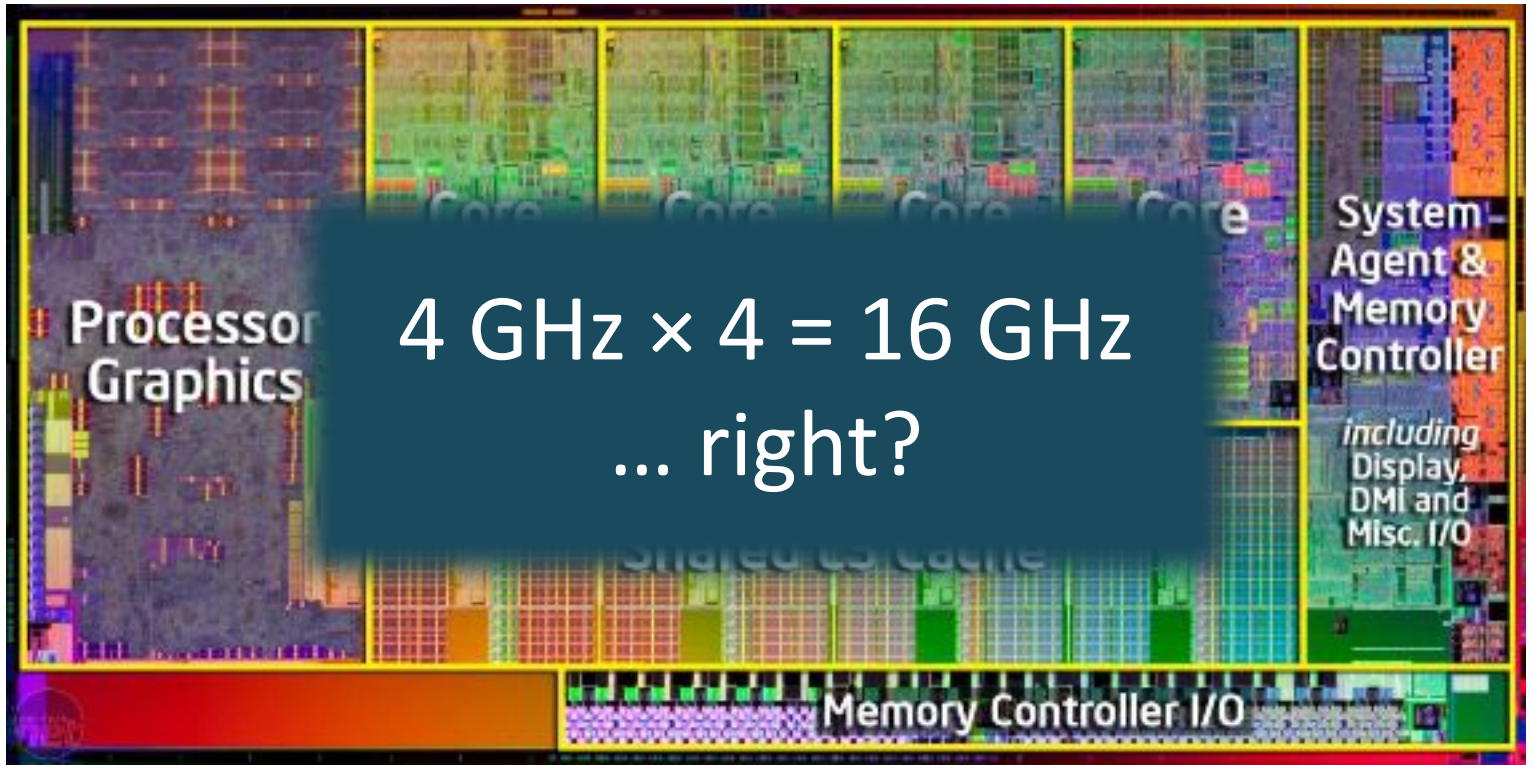
The number of transistors on a chip doubles
every 18-24 months.

(Historically has correlated with CPU clock speed)



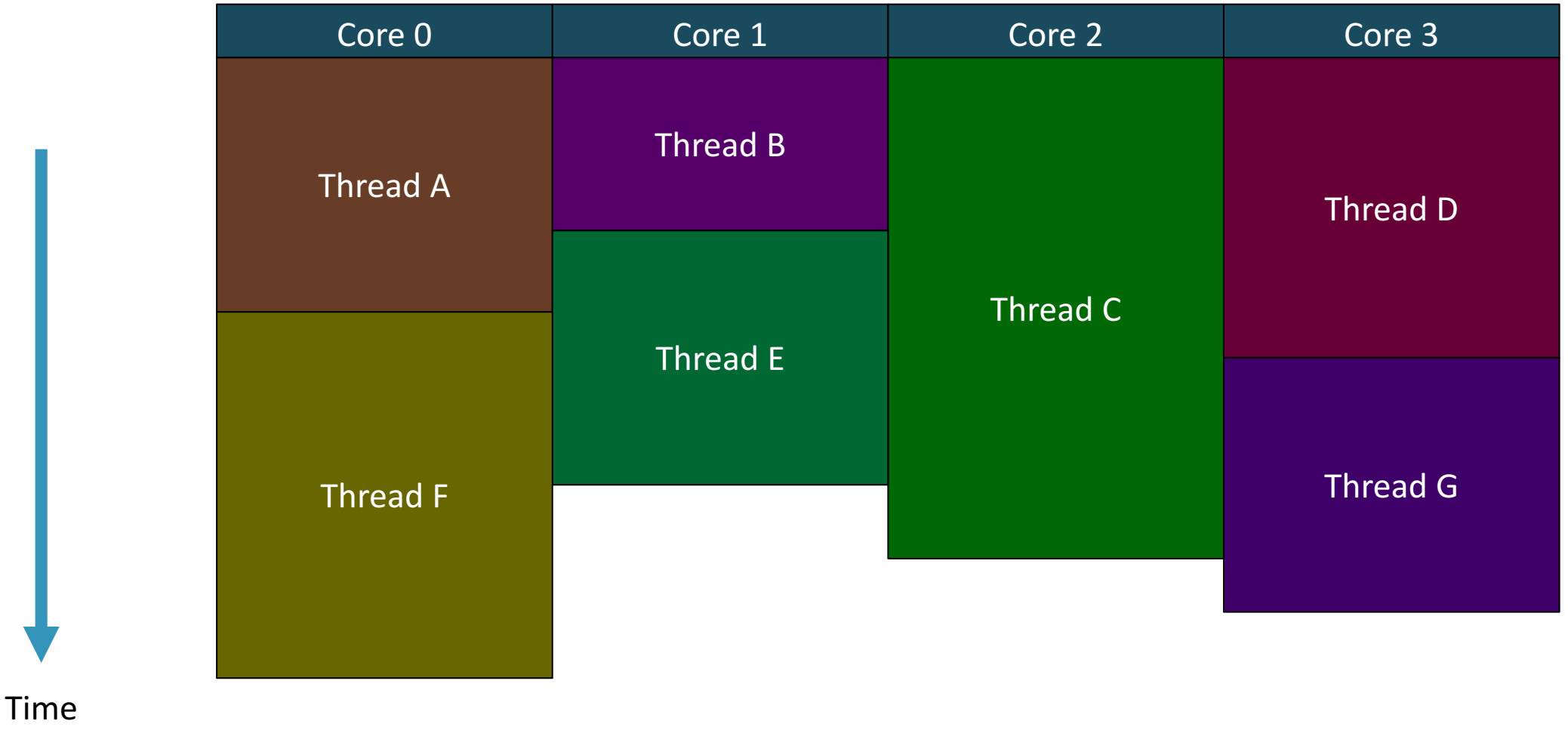
(adapted from Sutter 2009)

Quad-Core Processor

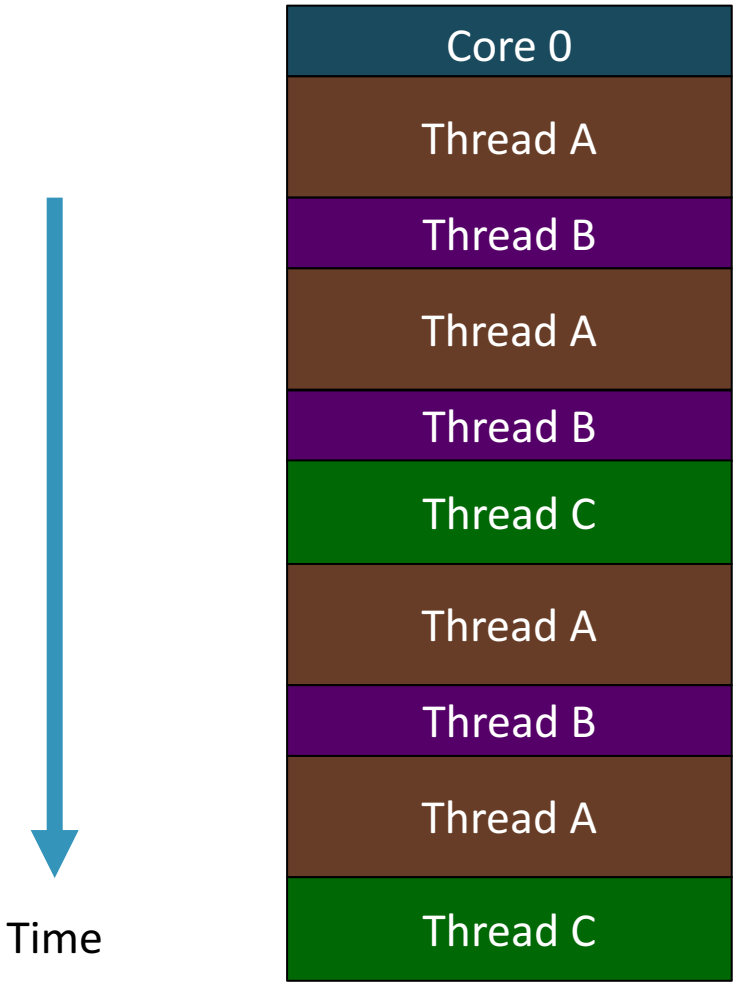


(CNET 2011)

Multi-Threading (on multiple cores)



Multi-Threading (time-slicing)

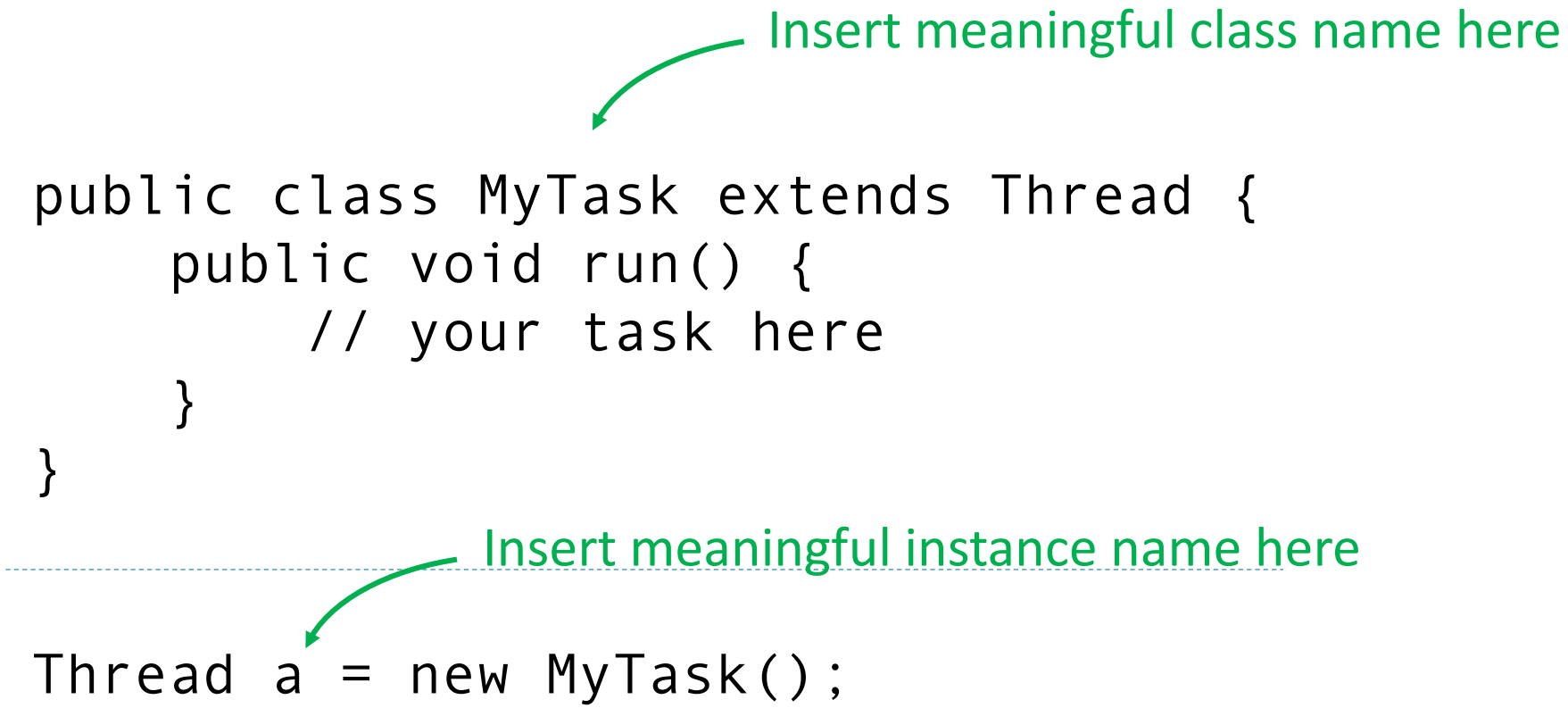


When there are more threads than cores:

The instructions are interleaved

Good for background garbage collection (GC) and responsive GUIs

Java Threads


Insert meaningful class name here
public class MyTask extends Thread {
public void run() {
// your task here
}
}
Insert meaningful instance name here
Thread a = new MyTask();

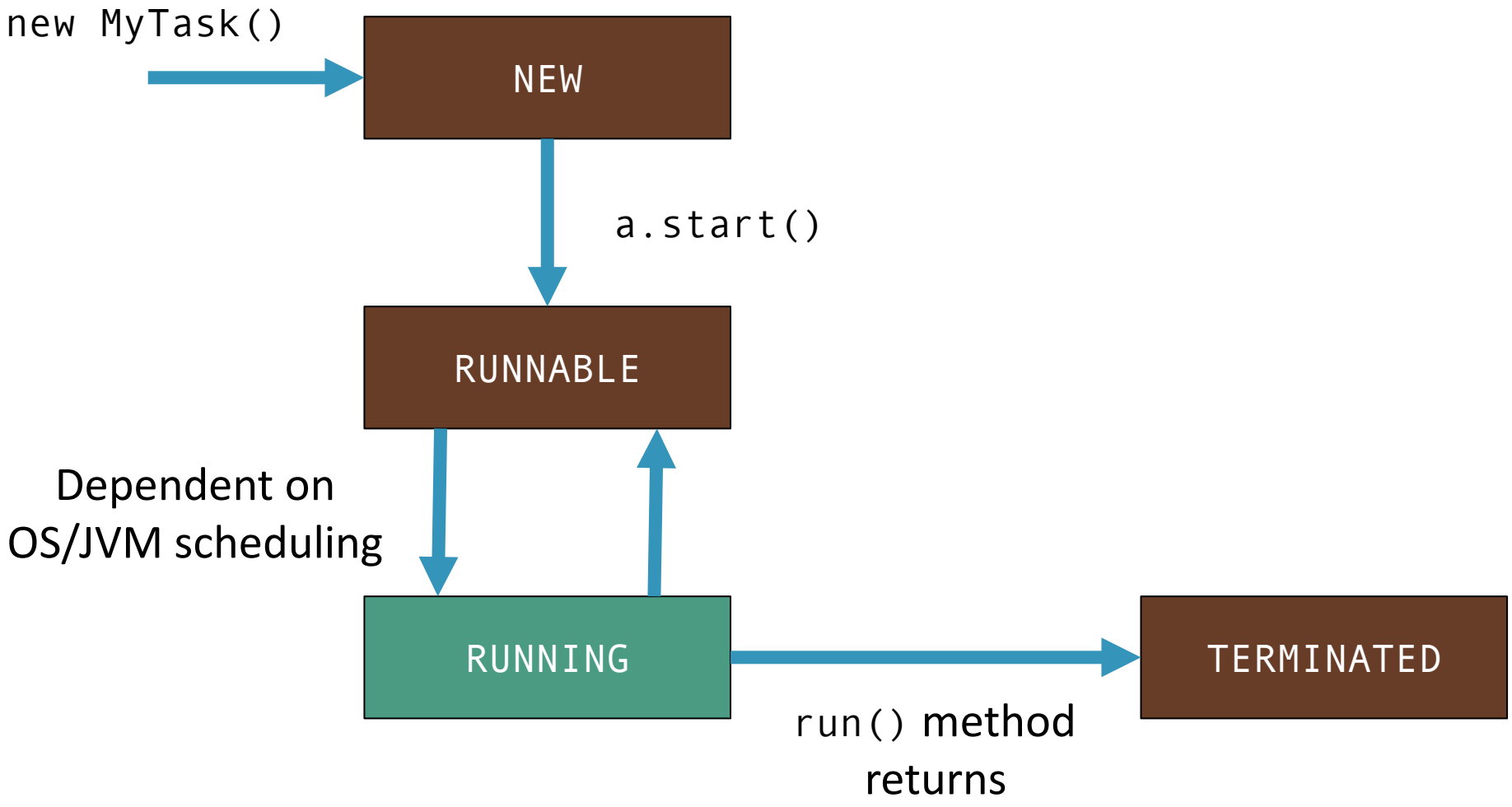
```
public class MyTask extends Thread {
    public void run() {
        // your task here
    }
}
```

```
Thread a = new MyTask();

a.run();
```

Invokes run() on the current thread.

```
public class MyTask extends Thread {
    public void run() {
        // your task here
    }
}
```

```
Thread a = new MyTask();

a.start();
```


new MyTask()
NEW
a.start()
RUNNABLE
Dependent on
OS/JVM scheduling
RUNNING
TERMINATED
run() method
returns

```
public class MyTask implements Runnable {
    public void run() {
        // your task here
    }
}
```

Alternate form

```
Runnable r = new MyTask();
Thread a = new Thread(r);

a.start();
```

Sharing Memory

Why?

Totally independent threads == separate processes

How?

Using the standard Java rules for variable scope / visibility

```
public static String name = ”Felix";

public class MyThread extends Thread {
    public void run() {
        System.out.println(name);
    }
}
```

Accessing name from thread “a”

```
Thread a = new MyThread();

a.start();

System.out.println(name);
```

Accessing name from main thread

```
public class Cat {
    private String name;
    public Cat(String name) {
        this.name = name;
    }
    public String getName() {
        return foo;
    }
}
```

```
public class MyThread extends Thread {
    private Holder holder;
    public MyThread(Holder holder) {
        this.holder = holder;
    }
    public void run() {
        System.out.println(name);
    }
}
```

```
Holder holder = new Holder();
Thread a = new MyThread();

a.start();

System.out.println(name);
```

In a real application, you would apply OO design principals such as encapsulation. However, the memory semantics are the same as accessing global variables.

```
String name = "Felix";
```

shared memory

```
System.out.println(name);
```

body of one thread

```
System.out.println(name);
```

body of another thread

We will use this layout to hide boilerplate code for the purposes of these examples.

```
int x = 0;
```

```
x++;
```

```
x++;
```

Q: What the value of x after this program runs?

A: either 1 or 2

```
int x = 0;
```

x++ does not map to a single CPU instruction.

```
LOAD x IN TO CPU REGISTER

INCREMENT CPU REGISTER
STORE FROM CPU REGISTER TO x
```

```
LOAD x IN TO CPU REGISTER
INCREMENT CPU REGISTER
STORE FROM CPU REGISTER TO x
```

```
int foo = 0;
```

```
foo = 42;
```

```
println(foo);
```

We don’t know which thread will run first.

Q: What does this program do?

A: prints either “0” or “42”

```
int foo = 0;
boolean done = false;
```

```
foo = 42;
done = true;
```

```
while (!done) {}
println(foo);
```

Q: What does this program do?

A: *Probably* prints “42” (maybe “0”, or just freezes)

Myth

Java runs exactly code you told it to,
in the order you wrote it.

Reality

The compiler / Java Virtual Machine (JVM) / CPU
may alter or re-order your code
for the purposes of performance optimization.

```
int foo = 0;
boolean done = false;
```

```
foo = 42;
done = true;
```

```
while (!done) {}
println(foo);
```

How might the optimiser break our program.

```
int foo = 0;
boolean done = false;
```

```
done = true;
foo = 42;
```

These two lines of code seem independent

Nobody will notice if they are re-ordered

```
while (!done) {}
println(foo);
```

How might the optimiser break our program.

```
int foo = 0;
boolean done = false;
```

The loop condition is not modified in the body of the loop

Now we can avoid the redundant re-checking.

```
done = true;
foo = 42;
```

```
if (!done) {
    while(true) {}
}
println(foo);
```

How might the optimiser break our program.

Definition : Data Race

- two or more threads in a single process access the same memory location concurrently, and
- at least one of the accesses is for writing, and
- the threads are not using any exclusive locks to control their accesses to that memory.

Result

the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order.

(Oracle)

Thread Safety 101 : Locks

```
final Object lock = new Object();
```

```
// code here

synchronized (lock) {
    // critical section
}

// more code
```

```
// code here

synchronized (lock) {
    // critical section
}

// more code
```

```
final Object lock = new Object();
```

```
// code here

synchronized (lock) {
    // critical section
}

// more code
```


```
// code here

synchronized (lock) {
    // critical section
}

// more code
```

```
final Object lock = new Object();
int x = 0;
```

```
synchronized (lock) {
    x++;
}
```


```
synchronized (lock) {
    x++;
}
```

Could be a larger block with many reads and writes to shared memory.

```
final Object lock = new Object();
```

When one thread unlocks, everything above that point "happens before" everything after a subsequent lock by another thread.

```
// code here

synchronized (lock) {
    // critical sect
}

// more code
```


```
// code here

synchronized (lock) {
    // critical section
}

// more code
```

```
final Object lock = new Object();
```

```
// code here

synchronized (lock) {
    // critical secti
}

// more code
```


```
// code here

synchronized (lock) {
    // critical section
}

// more code
```

We still don’t know which block will execute first.

```
final Object lock = new Object();
```

```
// code here

synchronized (lock) {
    // critical section
}

// more code
```

```
// code here

synchronized (lock) {
    // critical section
}

// more code
```

We know re-ordering won't move operations from inside-to-outside of the critical section.

```
int foo = 0;
boolean done = false;
final Object lock = new Object();
```

Excludes the updating thread

```
foo = 42;
synchronized (lock) {
    done = true;
}
```

```
synchronized (lock) {
    while (!done) {}
}
println(foo);
```

```
int foo = 0;
boolean done = false;
final Object lock = new Object();
```

```
foo = 42;
synchronized (lock) {
    done = true;
}
```


```
boolean ready = false;
while (!ready) {
    synchronized (lock) {
        ready = done;
    }
}
println(foo);
```

```
int foo = 0;
boolean done = false;
final Object lock = new Object();
```

```
foo = 42;
synchronized (lock) {
    done = true;
    lock.notify();
}
```


```
synchronized (lock) {
    while (!done) {
        lock.wait();
    }
}
println(foo);
```

Sharing Objects Safely

```
final Foo foo = new Foo();
```

```
if (foo.bing()) {
    foo.bar(4);
}
```

```
foo.bar(9);
int x = foo.baz();
for (Bar b : foo.bars()) {
    b.qux();
}
```

Q: Is this safe?

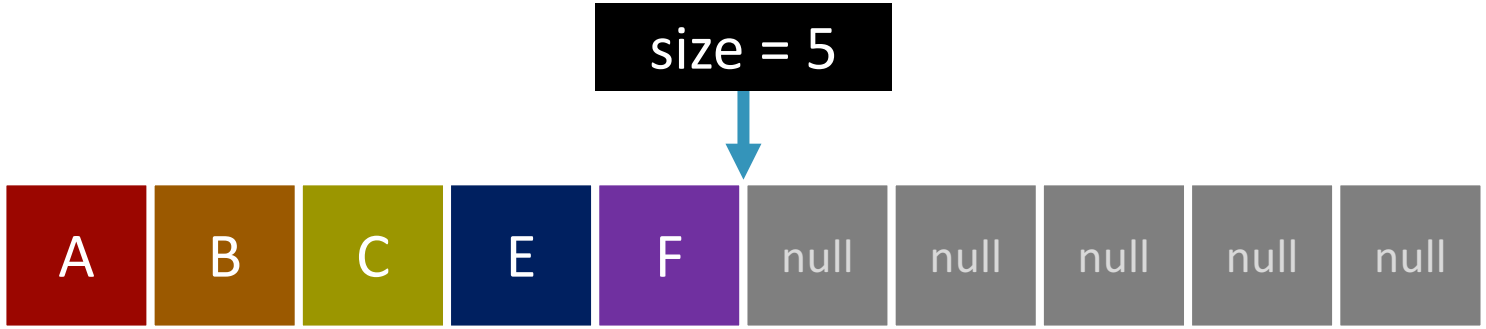
A: Read The Manual!

```
final List<String> list = new ArrayList<>();
```

```
list.add("Hello");
list.add("World");
list.remove(0);
```

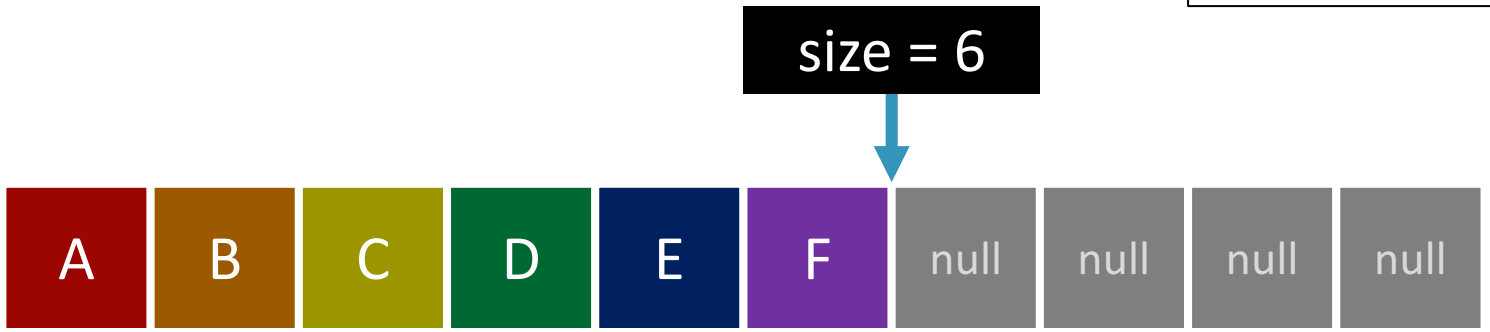
```
list.add("Java");
for (String x : list) {
    println(x);
}
```

Why might concurrent accesses to an object fail?

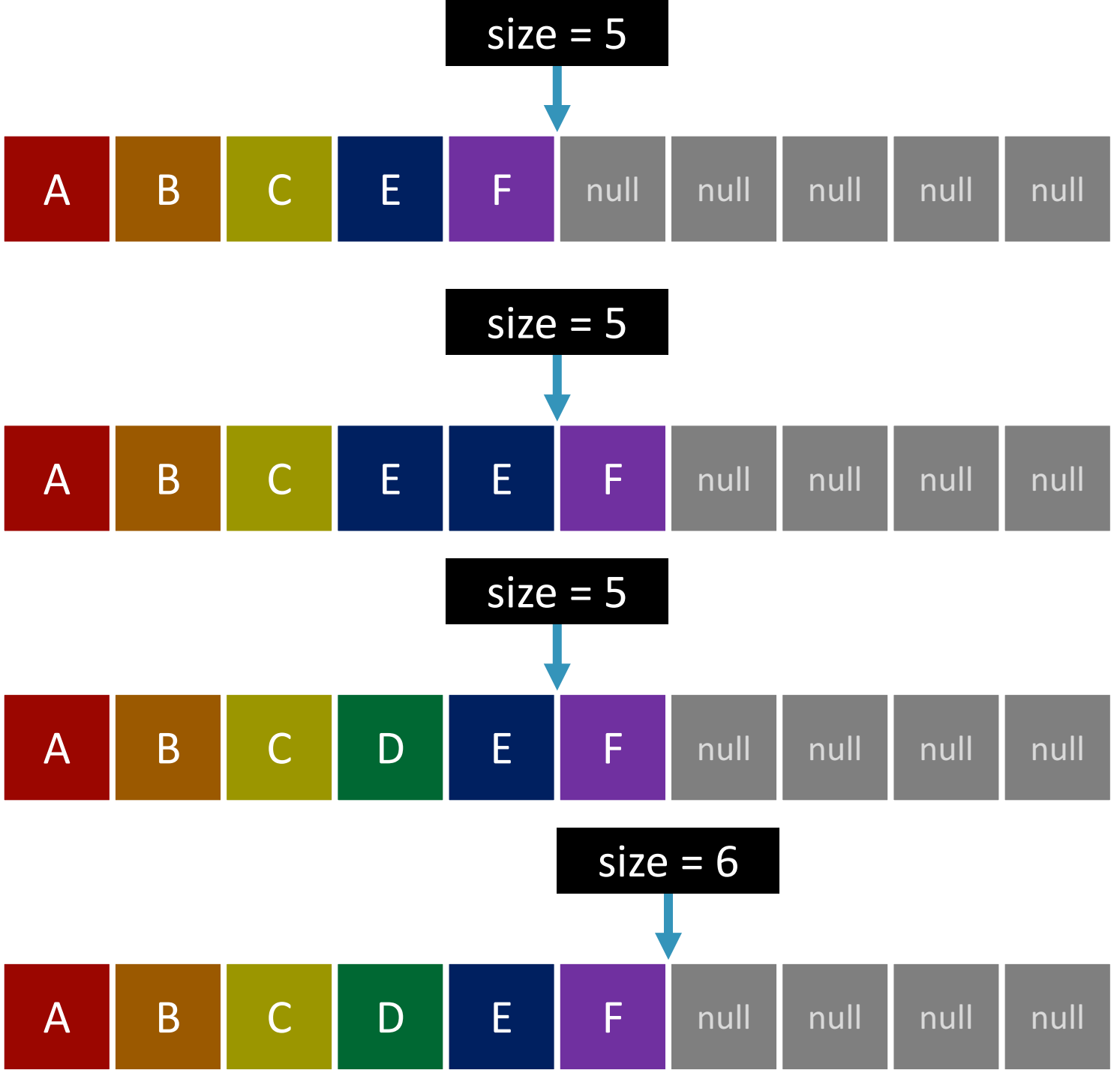



```
list.insert(3, "D");
```

From a single-threaded context, calling a method moves an object from one **valid** state to another **valid** state.



```
list.insert(3, "D");
```



Intermediate **invalid** states may be seen by another thread.

Different Levels of Thread Safety for Different Classes

Immutable	"… immutable object's externally visible state never changes, … it can never be observed to be in an inconsistent state …"	`String`
Thread-Safe	"… class's specification continue to hold when the object is accessed by multiple threads …"	`CopyOnWriteArrayList`
Conditionally Thread-Safe	"… each individual operation may be thread-safe, but certain sequences of operations may require external synchronization …"	`Vector`
Thread-Compatible	"… can be used safely in concurrent environments by using synchronization appropriately …"	`ArrayList`
Thread-Hostile	"… cannot be rendered safe to use concurrently …"	

(IBM developerWorks)

```
final List<String> list = new ArrayList<>();
final Object lock = new Object();
```

```
synchronized (lock) {
    list.add("Hello");
    list.add("World");
    list.remove(0);
}
```

```
synchronized (lock) {
    list.add("Java");
    for (String x : list) {
        println(x);
    }
}
```

ArrayList is thread compatible, so we can use it with locks.

```
final List<String> list = new Vector<>();
```

Lock on the list itself, not another lock.

```
list.add("Hello");
list.add("World");
list.remove(0);
```

```
list.add("Java");
synchronized (list) {
    for (String x : list) {
        println(x);
    }
}
```

According to the documentation for Vector, we only need to lock on iteration.

```
final List<String> list
        = Collections.synchronizedList(new ArrayList<>());
```

```
list.add("Hello");
list.add("World");
list.remove(0);
```

```
list.add("Java");
synchronized (list) {
    for (String x : list) {
        println(x);
    }
}
```

Vector is an older (almost-deprecated) class. We can get the same conditional thread safety using a wrapper on ArrayList.

```
final List<String> list = new CopyOnWriteArrayList<>();
```

```
list.add("Hello");
list.add("World");
list.remove(0);
```

```
list.add("Java");
for (String x : list) {
    println(x);
}
```

CopyOnWriteArrayList is thread-safe. When you iterate over the elements, you see a snapshot even if another thread modifies concurrently.

Can I safely concurrently do {some operations} with {some object}?

Step 1: Assume it’s not safe.

Step 2: Read the documentation.

Concurrency Utilities

```
final Random rnd = new Random();
```

```
double x = rnd.nextDouble();
```

```
double y = rnd.nextDouble();
```

Is it safe to concurrently use the default random number generator?

```
final Random rnd = new Random();
```

```
double x = rnd.nextDouble();
```

```
double y = rnd.nextDouble();
```

This is thread-safe. By why? Let's look inside the class.

```
long seed = initialize();
```

```
protected long next() {
    seed = f(seed);
    return g(seed);
}
```

This is the general structure of a random number generator.

```
long seed = initialize();
```

```
protected long next() {
    seed = f(seed);
    return g(seed);
}
```

Read followed by a write!

Not atomic!

Another read could modify the seed after we read by before we write. This is a data race.

```
long seed = initialize();
final Object lock = new Object();
```

```
protected long next() {
    synchronized (lock) {
        seed = f(seed);
    }
    return g(seed);
}
```

A second read, another thread may update first!

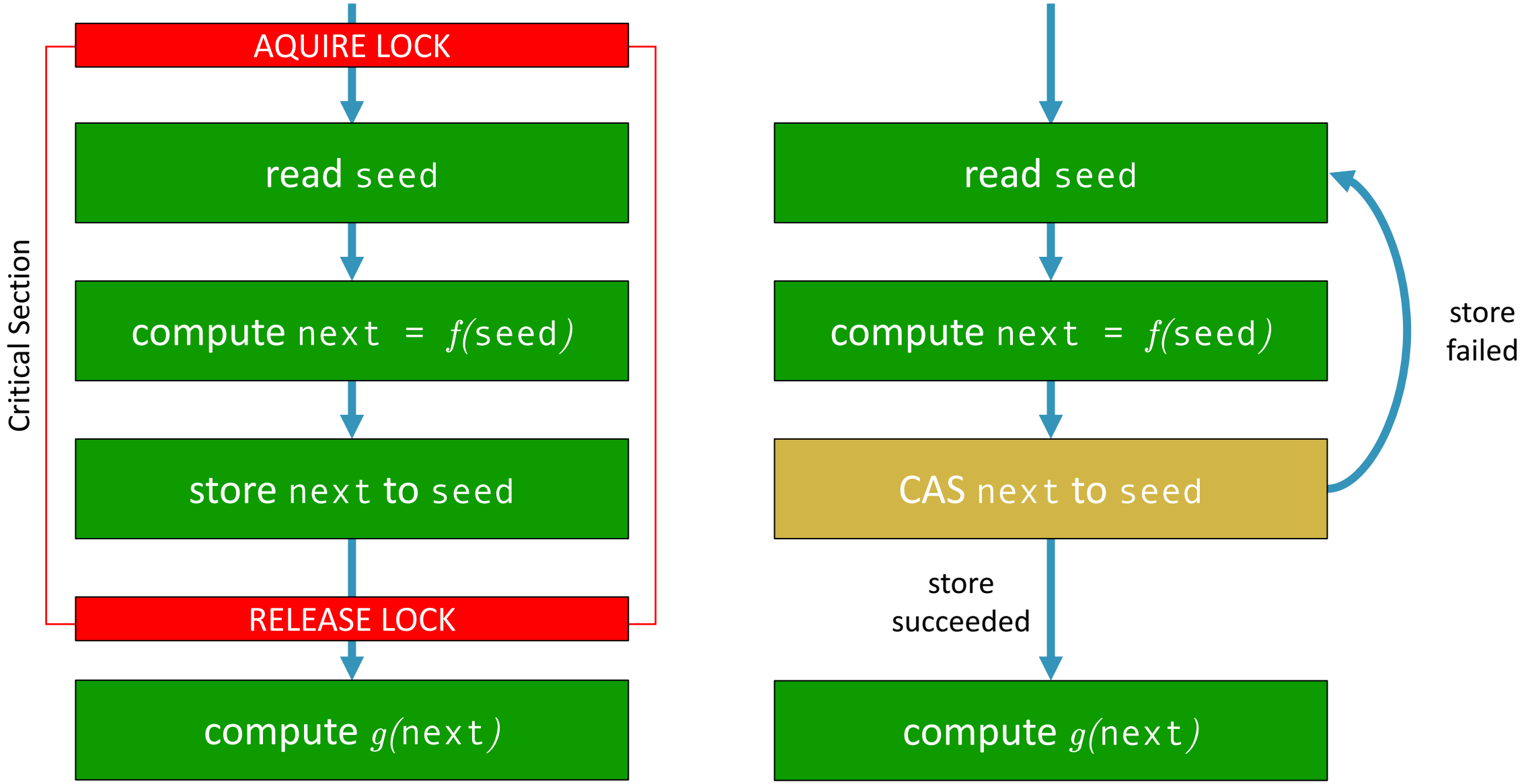
Another thread could modify before we read again. Another data race.

```
long seed = initialize();
final Object lock = new Object();
```

```
protected long next() {
    long next;
    synchronized (lock) {
        next = f(seed);
        seed = next;
    }
    return g(next);
}
```

This is how Java's Random actually worked in 1.0*

*Technically not line-for-line. Actually the whole method was locked, but this shows a lock around the critical section only.



"Pessimistic"

"Optimistic"

```
final AtomicLong seed = new AtomicLong(initialize());
```

```
protected long next() {
    long current, next;
    do {
        current = seed.get();
        next = f(current);
    } while (!seed.compareAndSet(current, next))
    return g(next);
}
```

We can do this optimistically with a compare-and-set operation.

```
final AtomicLong seed = new AtomicLong(initialize());
```

```
protected long next() {
    long current, next;
    do {
        current = seed.get();
        next = f(current);
    } while (!seed.compareAndSet(current, next))
    return g(next);
}
```

Remember the previous value.

```
final AtomicLong seed = new AtomicLong(initialize());
```

```
protected long next() {
    long current, next;
    do {
        current = seed.get();
        next = f(current);
    } while (!seed.compareAndSet(current, next))
    return g(next);
}
```

Compute the next value.

```
final AtomicLong seed = new AtomicLong(initialize());
```

```
protected long next() {
    long current, next;
    do {
        current = seed.get();
        next = f(current);
    } while (!seed.compareAndSet(current, next))
    return g(next);
}
```

Update only if another thread didn't get there first.

```
final AtomicLong seed = new AtomicLong(initialize());
```

```
protected long next() {
    long current, next;
    do {
        currenat = seed.get();
        next = f(current);
    } while (!seed.compareAndSet(current, next))
    return g(next);
}
```

Try again if we were sniped by another thread.

```
final AtomicLong seed = new AtomicLong(initialize());
```

```
protected long next() {
    long current, next;
    do {
        current = seed.get();
        next = f(current);
    } while (!seed.compareAndSet(current, next))
    return g(next);
}
```

Continue only after we succeed.

```
final AtomicLong seed = new AtomicLong(initialize());
```

```
protected long next() {
    long current, next;
    do {
        current = seed.get();
        next = f(current);
    } while (!seed.compareAndSet(current, next))
    return g(next);
}
```

This is how Java's Random works now!

Compare-and-set is a single instruction on your CPU. The Java JM uses native code to implement this.

```
final Random rnd = new Random();
```

```
double x = rnd.nextDouble();
```

If a lot of threads are using the random number generator often, threads will spin around the loop often.

✓ Single-threaded program

✓ Safe for multi-threading

✗ ~~Efficient when used by LOTS of threads~~

```
double x = ThreadLocalRandom.current().nextDouble();
```

ThreadLocalRandom generates a new Random for each thread. This is an example of thread-isolation.

✓ Single-threaded program
✓ Safe for multi-threading
✓ Efficient when used by LOTS of threads

ThreadLocalRandom uses the class ThreadLocal, which can be used to generate isolated instances of classes for different threads.

```
final AtomicLong x = new AtomicLong();
```

```
x.getAndIncrement();
```

```
x.getAndIncrement();
```

We can replace locks around x++ with CAS.

```
Map<String, Cat> cats = new HashMap<>();
```

```
String name = "Mittens";
synchronized (cats) {
  if (!cats.containsKey(name)) {
    cats.put(name, new Cat());
  }
}
```

```
synchronized (cats) {
  for (String name : cats.keySet() {
    println(name);
  }
}
```

What if we are doing more complex operations?

```
ConcurrentMap<String, Cat> cats = new ConcurrentHashMap<>();
```

```
String name = "Mittens";

cats.putIfAbsent(name, new Cat());
```

ConcurrentMap implementations provide atomic conditional operations.

```
for (String name : cats.keySet() {

    println(name);

}
```

Iterating over a ConcurrentMap will show a snapshot of the map.

java.util.concurrent

LinkedBlockingDeque

ArrayBlockingQueue

ConcurrentHashMap

ConcurrentSkipList

CopyOnWriteArrayList

Check out the concurrency utilities for concurrent collections, explicit locks, and atomics.

...

java.util.concurrent.locks

ReentrantLock

Condition

ReentrantReadWriteLock

...

java.util.concurrent.atomic

AtomicInteger

DoubleAccumulator

AtomicReference

...

Finally

Things to Remember

Watch out for data races

Use locks for safe sharing of memory

Read the manual for sharing objects

Use concurrency utilities to cut down on the need for client-side locking

In a Future Talk …

Functional programming in Java

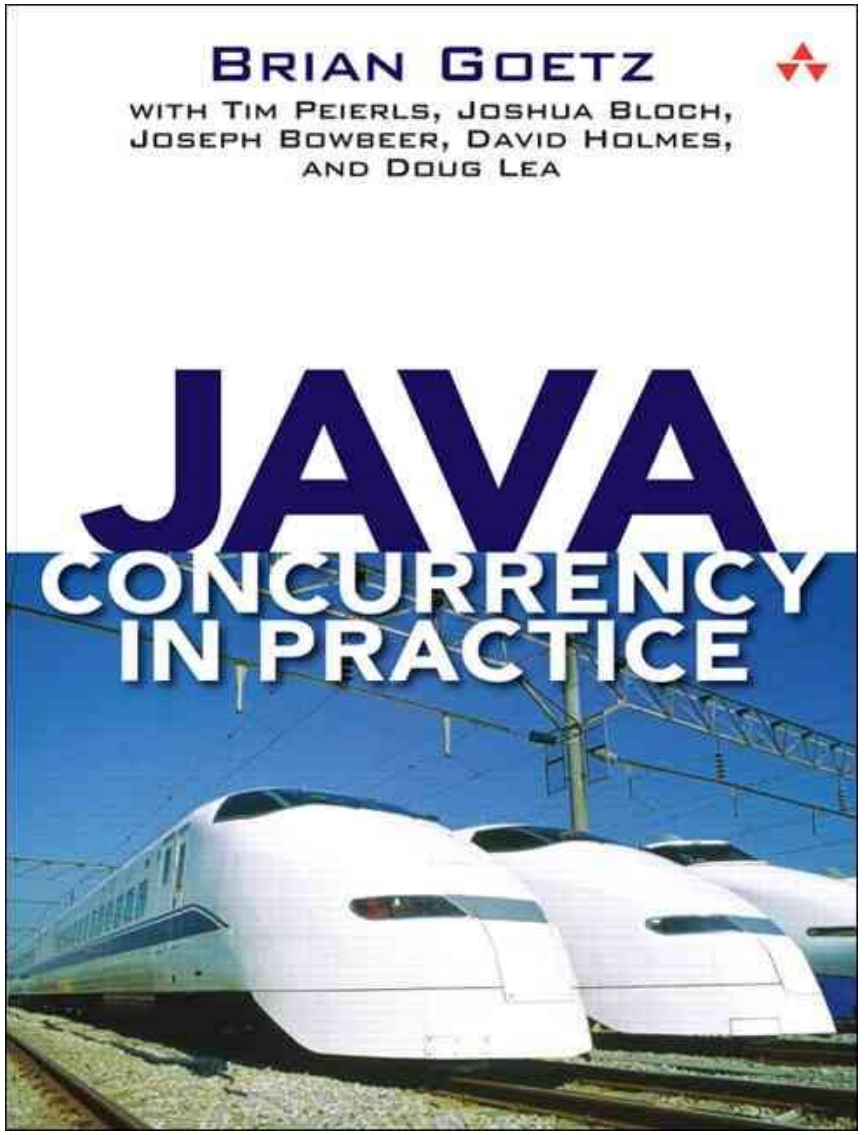
Abstracting away manual thread creation

Queuing work to be processed in the background

Processing data in highly parallel workflows

Further Reading

"Java Concurrency in Practice" (Goetz) covers a lot of practical advice for writing thread-safe code in Java.



Side References

(Sutter 2009)

http://www.gotw.ca/publications/concurrency-ddj.htm

(CNET 2011)

https://www.cnet.com/news/what-became-of-multi-core-programming-problems/

(Oracle)

https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html

(IBM developerWorks)

https://www.ibm.com/developerworks/library/j-jtp09263/index.html

Thank You

 @javaxnerd

 leechristie.com

 lee@leechristie.com