

Concurrency in weak memory models

Andreas Lööw

Chalmers, 5th year PhD student, formal methods
(= mathematical reasoning about software and hardware)

Aim of talk

Outline memory model related differences between programming in:

- “modelling languages” like pseudocode and Promela, and
- “real languages” like Java.

The talk is both Java specific and not Java specific:

- Java used as an example of a language with a “weak memory model”,
- but at least (subsets of) C and C++ similar

What to remember from this talk

In “modelling languages”, synchronization is used for:

- atomicity

In “real languages”, synchronization is used for:

- atomicity, and
- **visibility**

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model

What are memory models?

- Memory model part of language semantics (what programs mean)
- Different memory models exist
- In pseudocode, **sequential consistency (SC)** often assumed -- one of the "strongest" memory models
- Java, instead, offers the **Java memory model (JMM)**, one particular "weak" memory model

OK... but what is a memory model?

- More or less: Semantics of shared variables (and synchronization)
- Consider the question: What values are variable reads allowed to return?
- ???

Reading variables: Sequential programming

Obvious answer: The **latest** value we wrote to the variable

```
int x = 0, y = 0;
```

```
x = 1;
```

```
y = 1;
```

```
print(y); // will obviously print 1
```

```
print(x); // again, prints 1
```


Reading variables: Concurrent programming

```
int done = false, r = 0;
```

```
t1 {  
    r = 666;  
    done = true;  
}
```

```
t2 {  
    if (done)  
        print(r);  
}
```

Assuming sequential consistency:
Just consider all possible
interleavings!

Reading variables: Sequential consistency (SC)

```
int done = false, r = 0;
```

```
done?  
r = 666;  
done = true;
```

Output:

-

```
r = 666;  
done?  
done = true;
```

Output:

-

```
r = 666;  
done = true;  
done?  
print(r);
```

Output:

666

Reading variables: The Java memory model

```
int done = false, r = 0;
```

```
t1 {  
    r = 666;  
    done = true;  
}
```

```
t2 {  
    if (done)  
        print(r);  
}
```

Assuming the Java memory model:
What output can we see now?

- a) Same as before: - or 666
- b) -, 0, 666
- c) This is undefined behavior, so anything can be printed

Demo OutOfOrderTest.java

Reading variables: Sequential consistency (SC)

```
int done = false, r = 0;
```

```
t1 {  
    r = 666;  
    done = true;  
}
```

```
t2 {  
    if (done)  
        print(r);  
}
```

Some visibility guarantees in SC:

- "Program order" always maintained
 - In particular, `r = 666` always before `done = true` in any interleaving
- No "stale" values: Always see the latest value written to any variable
- But the above guarantees not provided by some weak memory models!

Reading variables: Weak memory models

```
int done = false, r = 0;
```

```
t1 {  
    r = 666;  
    done = true;  
}
```

```
t2 {  
    if (done)  
        print(r);  
}
```

”Interleaving-based semantics” in some sense the ”obvious” semantics for concurrency

Why make things more difficult? Why give up program order and other nice things?

Because: SC costs too much

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model

SC cost 1: Prohibits (too many) compiler optimizations

- Aaaaah!!! Messiness! Real-world things! In pseudocode we do not have to consider ugliness such as compiler "details" etc.
- Example: For some compiler optimizations we want to reorder writes to variables. (For whatever reason: Might improve register allocation or anything.)

SC cost 1: Prohibits (too many) compiler optimizations

- E.g., the transformation to the right “semantics preserving” in sequential setting if we only consider final state of program
- Not equivalent if we can inspect program under execution, which we can if x and y are shared variables in a concurrent setting
- Breaks illusion of “program order”!

Original program:

```
x = 1;
```

```
y = 2;
```

```
z = x + y; // x = 1, y = 2, z = 3
```

Transformed program:

```
y = 2;
```

```
x = 1;
```

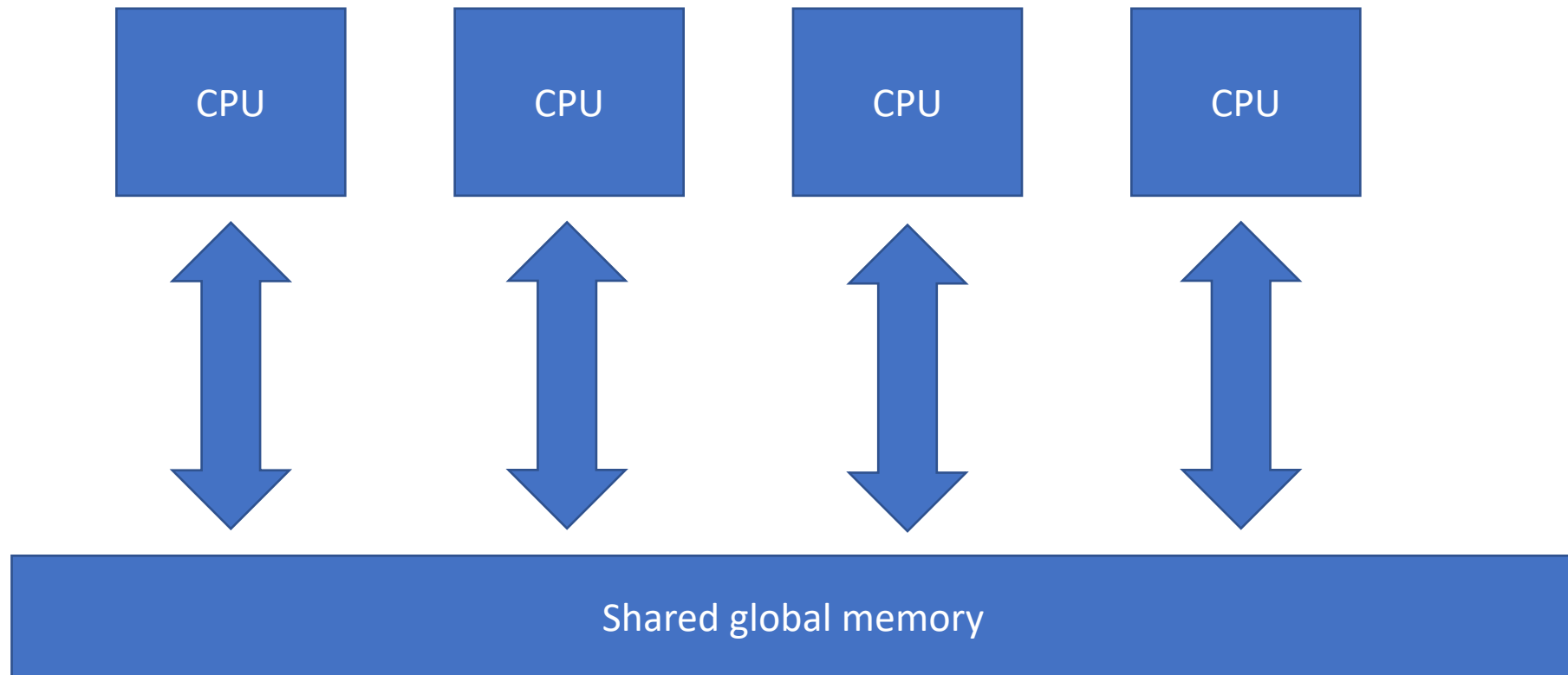
```
z = x + y; // x = 1, y = 2, z = 3
```



Write order swapped

SC cost 2: Causes too much cache synchronization

Cost of SC not obvious with too simplified machine models:



SC cost 2: Causes too much cache synchronization

More realistic

Btw, modern CPUs execute instructions out-of-order and in parallel (which can also break illusion of program order)

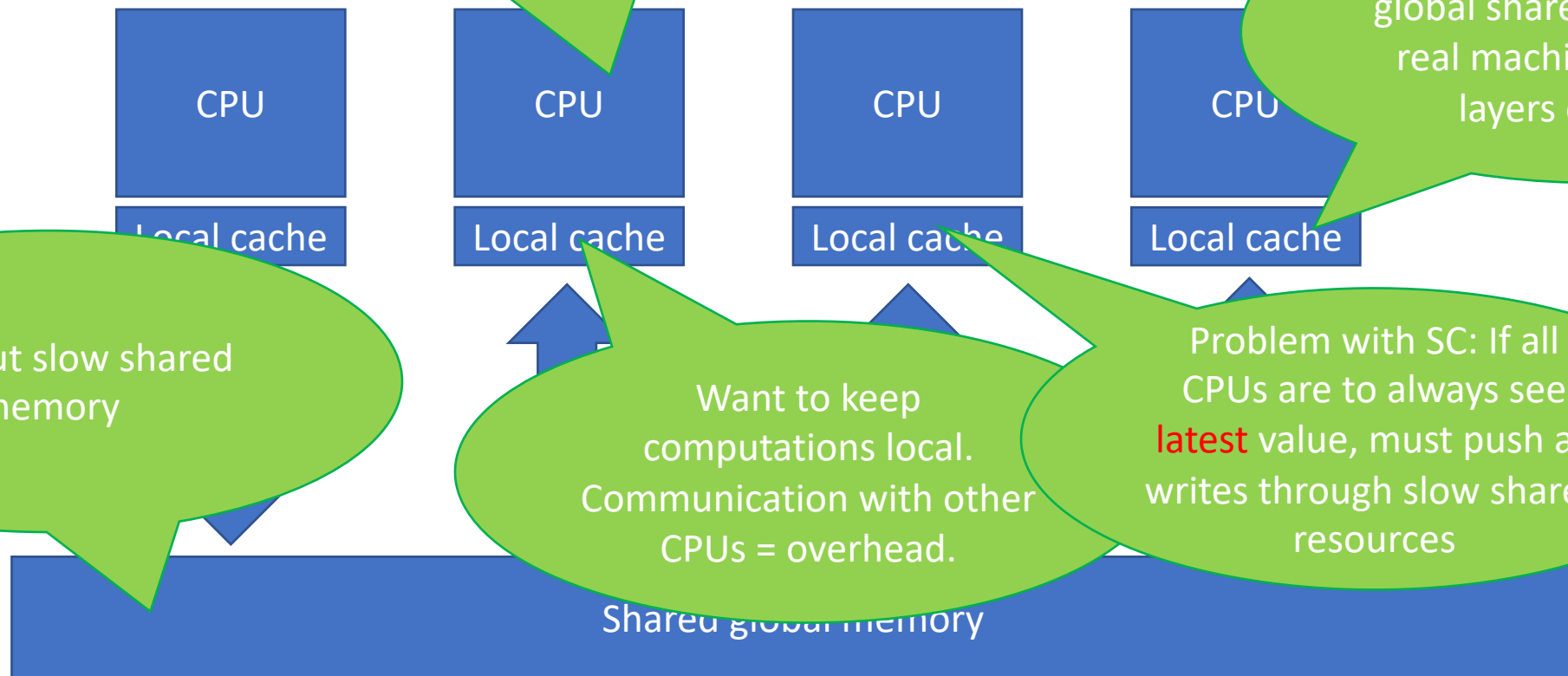
(Realistic) model of today's computers:

Small but fast compared to global shared memory. (In real machines: multiple layers of cache.)

Large but slow shared memory

Want to keep computations local. Communication with other CPUs = overhead.

Problem with SC: If all CPUs are to always see **latest** value, must push all writes through slow shared resources



Shared global memory

Why not SC: Summary

- Not a complete list of reasons, just two examples!
- Anyhow, in summary:
SC too expensive in many situations
- Solution to mentioned problems:
Relax some guarantees offered by SC → we get weak memory models
- Weaker memory models (potentially) **more performant**, but **more difficult to program in**

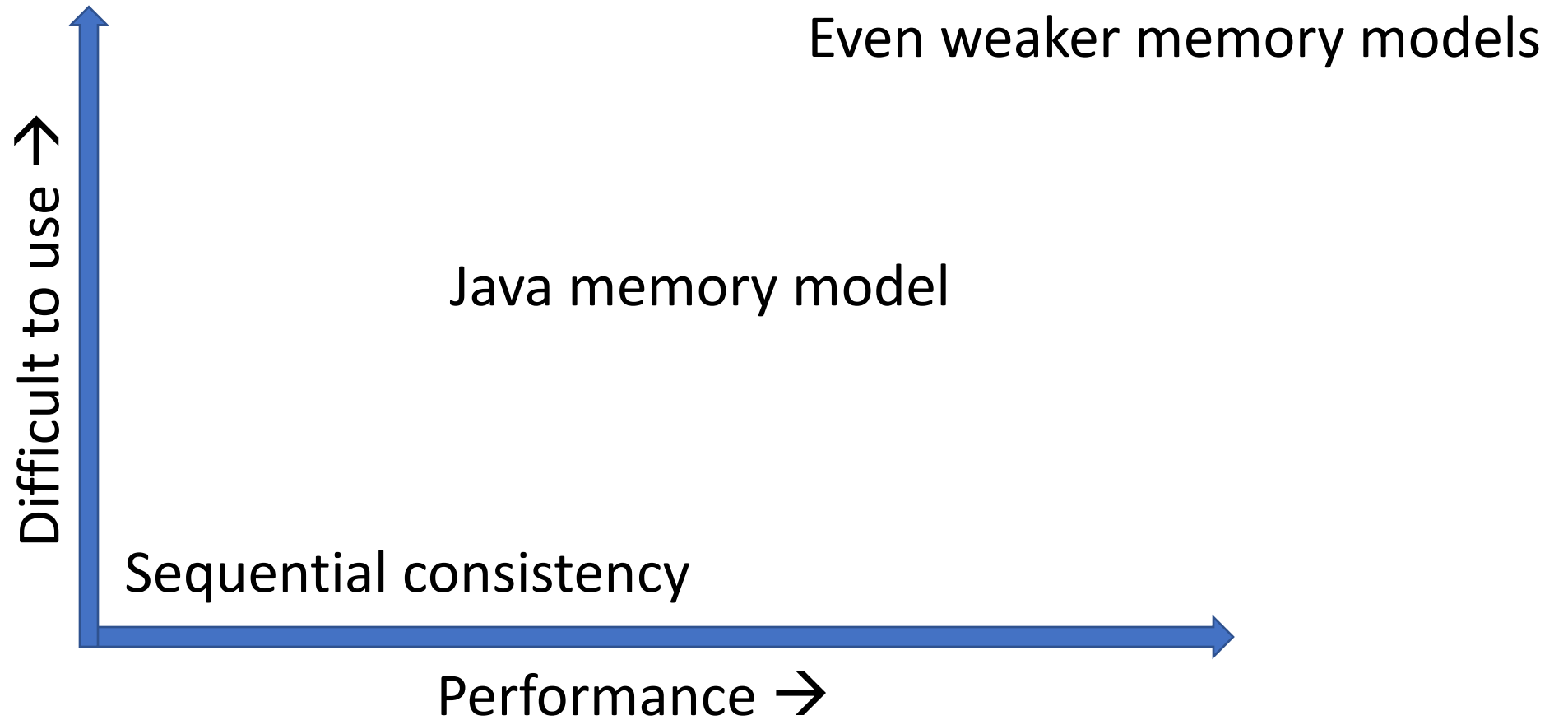
Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model (as an example of programming in a weak memory model)

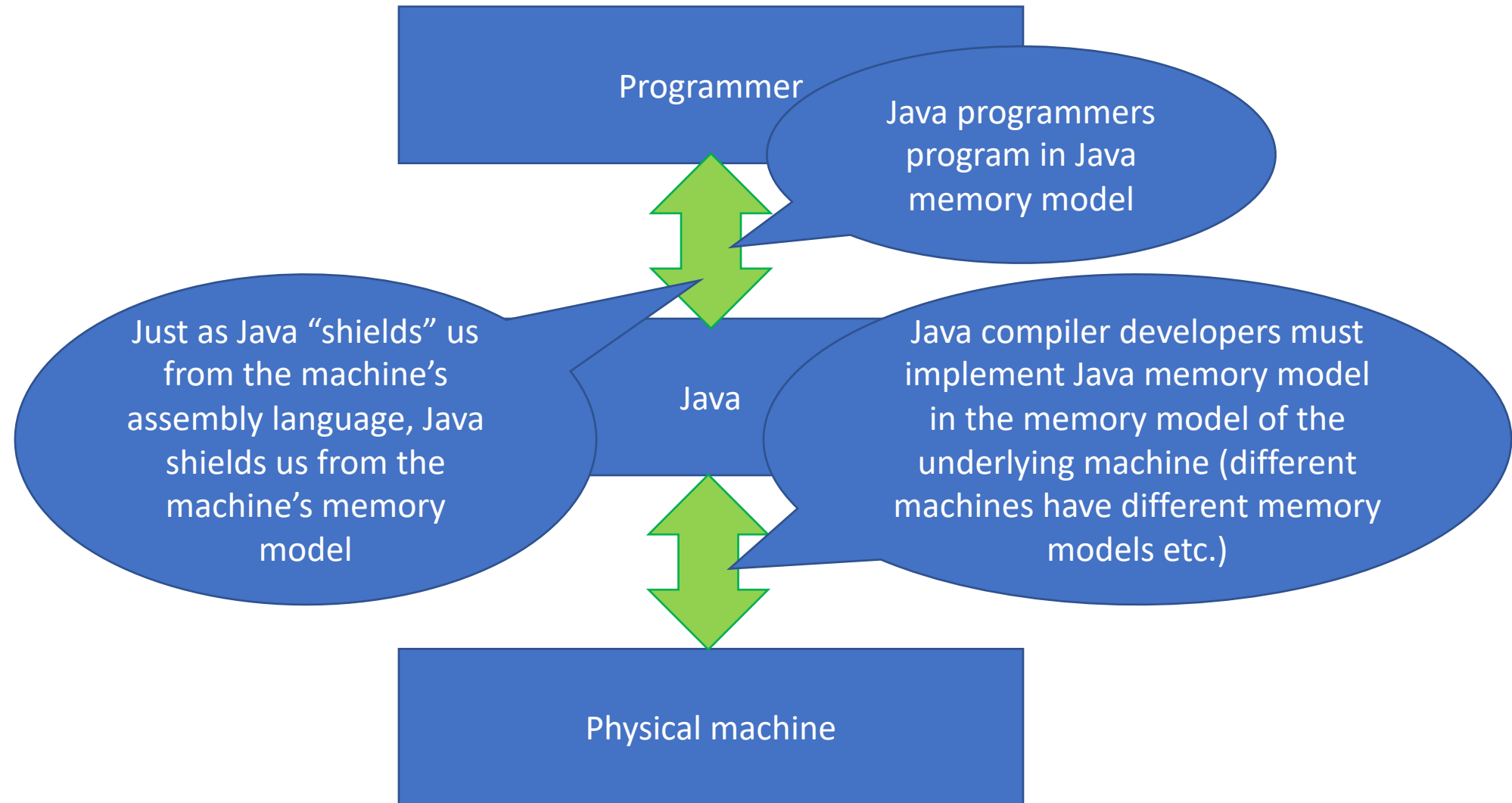
The Java memory model

- Less convenient than SC, but implementable on modern machine architectures without too much performance loss
- Opinion: Memory model part of language design, and different coordinates in the design space have different tradeoffs. As with any other language feature: No “right” answer.

Design tradeoff space



More context: Some more machine details



SC for data-race-free programs

- A few (C-like) languages have converged to “**sequential consistency for data-race-free programs**” memory models
- Java included in this family
- Reasoning principle: **If there are no data races (under SC), we can assume SC when reasoning about our program**
- Important to remember definitions of **data race** and **race conditions** (and not mix them up)

Data races

Slight variation of previous definition you seen, to fit Java better:

Def. Two memory accesses are in a data race iff

- they access the same **memory location simultaneously** (they are interleaved next to each other),
- at least one access is a **write**,
- **insufficient explicit synchronization** used to protect the accesses

Def. A program is data-race-free iff no SC execution of the program contain a data race.

“Slight variation”? Note that we quantify over all SC executions in the second definition.

Note that data-race-freedom is a “**language-level**” property!

Definition of data race surprisingly subtle

E.g., does this program contain any data races?

```
bool x = false, y = false;
```

```
t1 {  
    if (x) y = true;  
}
```

```
t2 {  
    if (y) x = true;  
}
```

No!

Race conditions

Definition from course slides:

Def. A *race condition* is a situation where the **correctness** of a concurrent program depends on the specific execution.

Note that this is an "**application-level**" property!

I.e., for a given program p , to answer the question "is p free from race conditions?" we must have access to the specification of p .

SC for data-race-free programs, again

- For Java programs, we have SC for programs **without data races**
- Presence of race conditions does not rob us of SC – important to know (the difference between) the two definitions
- What about the semantics of programs *with* data races?
 - Will not be considered here
 - In e.g. C++ data races result in undefined behavior (see C++ specification or https://en.cppreference.com/w/cpp/language/memory_model)
 - Java is supposed to be a "safe language", some guarantees (e.g. out-of-thin-air safety)

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model (as an example of programming in a weak memory model)

Practice?

- But what does this mean in practice?
- I.e: How does “weak memory models” affect my daily life as a programmer?
- Answer: You must “annotate” your program more (compared to SC). “Annotations” in the form of variable qualifiers, synchronization mechanisms etc.
- Essentially annotating which things are shared and which are not

Simple example

Simpler than initial example,
only one variable here

```
bool done = false;
```

```
t1 {  
    done = true;  
}
```

```
t2 {  
    if (done) print(33);  
}
```

- Does this program contain
 - data races?
 - race conditions?
- Data race = yes, done is accessed without synchronization and one of the accesses is a write
- Race condition = depends on the specification we are to satisfy (what it means for the program to be correct)
- (Note: Difficult to reason about race conditions (correctness) because we cannot assume SC because we have data races!)

Simple example

```
bool done = false;
```

```
t1 {  
    done = true;  
}
```

```
t2 {  
    if (done) print(33);  
}
```

- Wait a minute!
- Are you telling me there's a problem in this program?
- From a SC perspective, everything is fine!
- No atomicity problems or anything like that... but **visibility** problems!

Simple example (fixed)

```
volatile bool done = false;
```

- Solution: Annotate your program. E.g., in Java `volatile` is considered synchronization.

```
t1 {  
    done = true;  
}
```

- Does this program contain
 - data races?
 - race conditions?

```
t2 {  
    if (done) print(33);  
}
```

- Data race = no, in Java `volatile` accesses are considered synchronized
- Race condition = ???, still depends on specification

Language
dependent

Simple example (fixed)

`volatile` bool done = false; Example specification:

```
t1 {  
    done = true;  
}
```

- Spec = “If the program outputs something, it must output 33”

- (In other words: Spec = “Output nothing or 33”)

```
t2 {  
    if (done) print(33);  
}
```

- Race conditions w.r.t. above specification?

- No race conditions! (As correct output does not depend on specific “execution”/ interleaving.)

Simple example (fixed)

`volatile` bool done = false; Another example specification:

```
t1 {  
    done = true;  
}
```

- Spec = “The program outputs 33”

- Race conditions w.r.t. above specification?

```
t2 {  
    if (done) print(33);  
}
```

- Yes, have race condition. Some interleavings give us correct output, others do not.

Similar example, with locks

```
lock lock = new lock();  
int id = 0;
```

```
t1 {  
    lock.lock();  
    id++;  
    lock.unlock();  
}
```

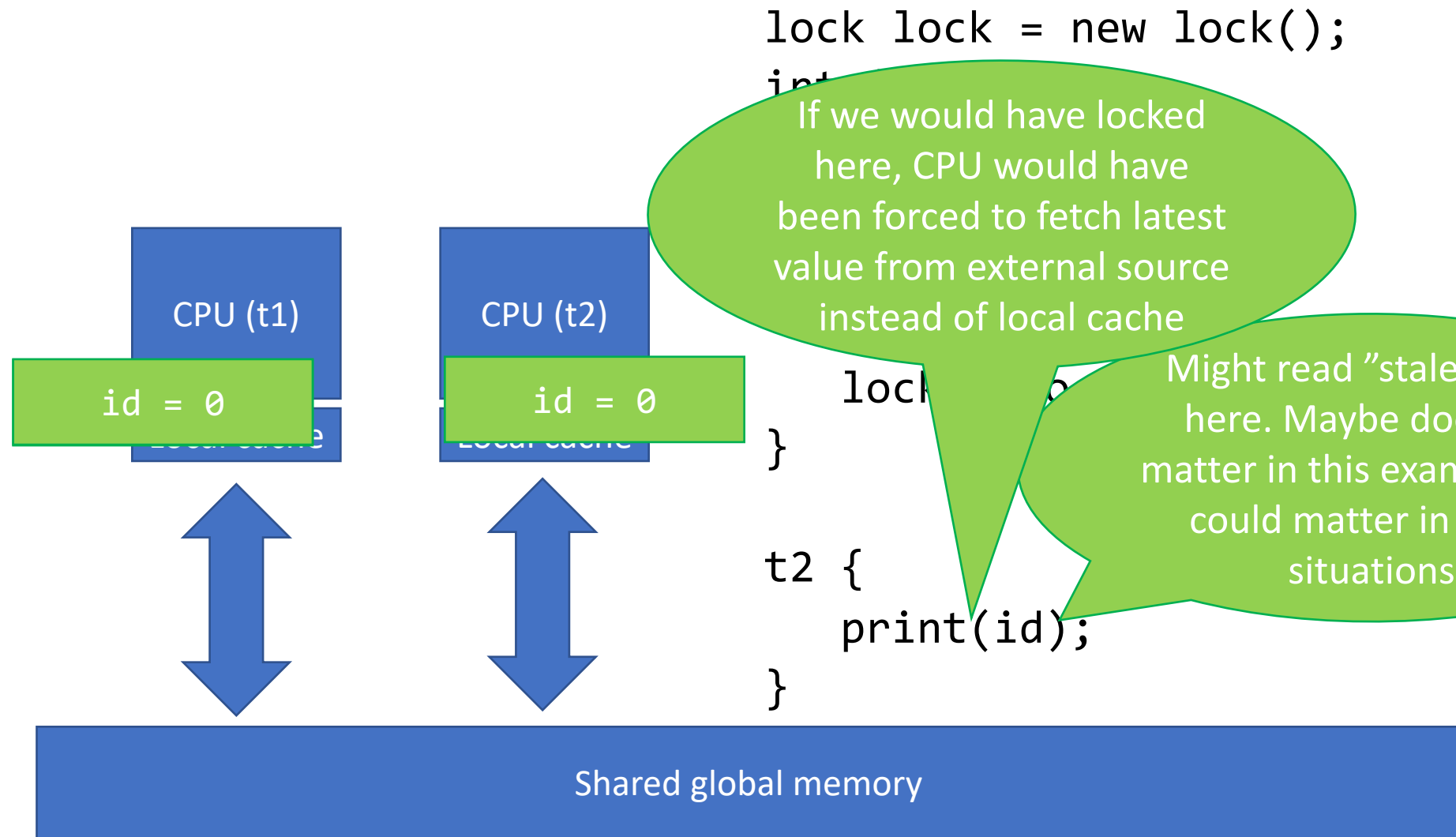
```
t2 {  
    print(id);  
}
```

Data races?

We have a race! All accesses to the shared variable done must be synchronized!

Here we have (again) atomicity, but not:
visibility

id flag might exist as multiple copies...



NOTE: Everything on this slide simplified, and makes unsound assumptions about JVM implementation details

Similar example, with locks (fixed)

```
lock lock = new lock();  
int id = 0;
```

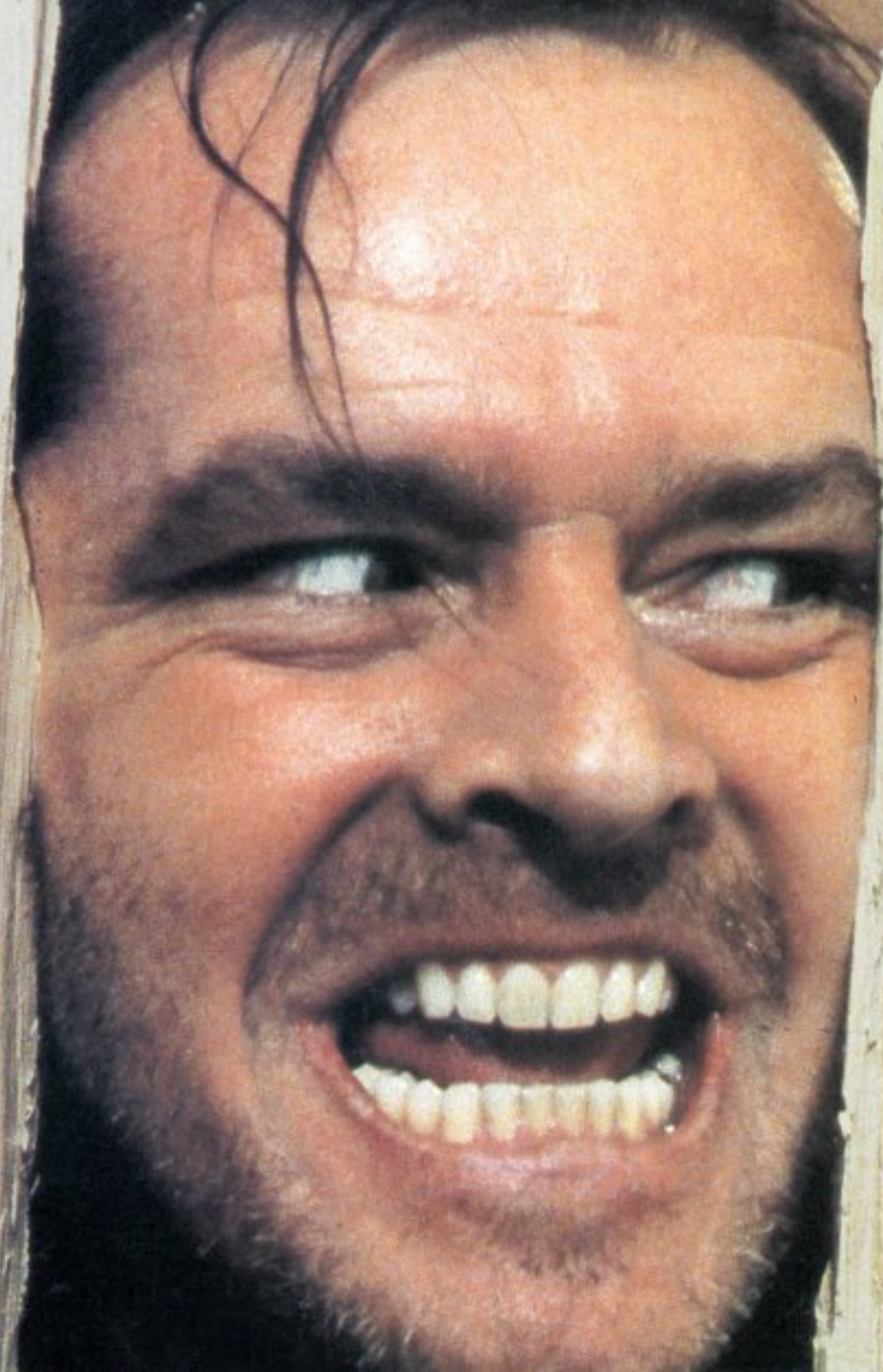
```
t1 {  
    lock.lock();  
    id++;  
    lock.unlock();  
}
```

```
t2 {  
    lock.lock(); // new  
    print(id);  
    lock.unlock(); // new  
}
```

This is how the program would look like with proper annotations/synchronization

No data races in sight!

More
details on
the Java
memory
model



Module [java.base](#)

Package [java.util.concurrent](#)

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the [java.util.concurrent.locks](#) and [java.util.concurrent.atomic](#) packages.

Executors

Interfaces. [Executor](#) is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous I/O, and lightweight task frameworks. Depending on which concrete [Executor](#) class is being used, tasks may execute in a newly created thread, an existing task-execution thread, or the thread calling [execute](#), and may execute sequentially or concurrently. [ExecutorService](#) provides a more complete asynchronous task execution framework. An [ExecutorService](#) manages queuing and scheduling of tasks, and allows controlled shutdown. The [ScheduledExecutorService](#) subinterface and associated interfaces add support for delayed and periodic task execution. [ExecutorServices](#) provide methods arranging asynchronous execution of any function expressed as [Callable](#), the result-bearing analog of [Runnable](#). A [Future](#) returns the results of a function, allows determination of whether execution has completed, and provides a means to cancel execution. A [RunnableFuture](#) is a [Future](#) that possesses a [run](#) method that upon execution, sets its results.

Implementations. Classes [ThreadPoolExecutor](#) and [ScheduledThreadPoolExecutor](#) provide tunable, flexible thread pools. The [Executors](#) class provides factory methods for the most common kinds and configurations of [Executors](#), as well as a few utility methods for using them. Other utilities based on [Executors](#) include the concrete class [FutureTask](#) providing a common extensible implementation of [Futures](#), and [ExecutorCompletionService](#), that assists in coordinating the processing of groups of asynchronous tasks.

Class [ForkJoinPool](#) provides an [Executor](#) primarily designed for processing instances of [ForkJoinTask](#) and its subclasses. These classes employ a work-stealing scheduler that attains high throughput for tasks conforming to restrictions that often hold in computation-intensive parallel processing.

Queues

- they are guaranteed to traverse elements and modifications subsequent to construction.

We can also say “memory consistency model”

guaranteed to) reflect any

Memory Consistency Properties

Chapter 17 of *The Java Language Specification* defines the *happens-before* relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be *visible* to a read by another thread only if the write operation *happens-before* the read operation. The `synchronized` and `volatile` constructs, as well as the `Thread.start()` and `Thread.join()` methods, can form *happens-before* relationships. In particular:

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order.
- An unlock (synchronized block or method exit) of a monitor *happens-before* every subsequent lock (synchronized block or method entry) of that same monitor. And because the *happens-before* relation is transitive, all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that monitor.
- A write to a `volatile` field *happens-before* every subsequent read of that same field. Writes and reads of `volatile` fields have similar memory consistency effects as entering and exiting monitors, but do *not* entail mutual exclusion locking.
- A call to `start` on a thread *happens-before* any action in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a `join` on that thread.

The methods of all classes in `java.util.concurrent` and its subpackages extend these guarantees to higher-level synchronization. In particular:

- Actions in a thread prior to placing an object into any concurrent collection *happen-before* actions subsequent to the access or removal of that element from the collection in another thread.
- Actions in a thread prior to the submission of a `Runnable` to an `Executor` *happen-before* its execution begins. Similarly for `Callable`s submitted to an `ExecutorService`.
- Actions taken by the asynchronous computation represented by a `Future` *happen-before* actions subsequent to the retrieval of the result via `Future.get()` in another thread.
- Actions prior to "releasing" synchronizer methods such as `Lock.unlock`, `Semaphore.release`, and `CountDownLatch.countDown` *happen-before* actions subsequent to a successful "acquiring" method such as `Lock.lock`, `Semaphore.acquire`, `Condition.await`, and `CountDownLatch.await` on the same synchronizer object in another thread.
- For each pair of threads that successfully exchange objects via an `Exchanger`, actions prior to the `exchange()` in each thread *happen-before* those subsequent to the corresponding `exchange()` in another thread.
- Actions prior to calling `CyclicBarrier.await` and `Phaser.awaitAdvance` (as well as its variants) *happen-before* actions performed by the barrier action, and actions performed by the barrier action *happen-before* actions subsequent to a successful return from the corresponding `await` in other threads.

Happens-before example

```
static int x = 1;
```

```
x = 2;
```

```
// What can be printed?
```

```
Thread t = new Thread(() ->
```

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

```
t.start();
```

- Data race because t reads x without synchronization?

- (Could potentially argue read and write not overlapping in any SC execution.)

- *x write happens-before x read, because happens-before transitive*

- they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.

Memory Consistency Properties

Chapter 17 of *The Java Language Specification* defines the *happens-before* relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation *happens-before* the read operation. The synchronized and volatile constructs, as well as the `Thread.start()` and `Thread.join()` methods, can form *happens-before* relationships. In particular:

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order.
 - An unlock (synchronized block or method exit) of a monitor *happens-before* every subsequent lock (synchronized block or method entry) of that same monitor. And because the *happens-before* relation is transitive, all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that monitor.
- A write to a volatile field *happens-before* every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do *not* entail mutual exclusion locking.
 - A call to `start` on a thread *happens-before* any action in the started thread.
 - All actions in a thread *happen-before* any other thread successfully returns from a `join` on that thread.

The methods of all classes in `java.util.concurrent` and its subpackages extend these guarantees to **higher-level synchronization**. In particular:

- Actions in a thread prior to placing an object into any concurrent collection *happen-before* actions subsequent to the access or removal of that element from the collection in another thread.
- Actions in a thread prior to the submission of a `Runnable` to an `Executor` *happen-before* its execution begins. Similarly for `Callable`s submitted to an `ExecutorService`.
- Actions taken by the asynchronous computation represented by a `Future` *happen-before* actions subsequent to the retrieval of the result via `Future.get()` in another thread.
- Actions prior to "releasing" synchronizer methods such as `Lock.unlock`, `Semaphore.release`, and `CountDownLatch.countDown` *happen-before* actions subsequent to a successful "acquiring" method such as `Lock.lock`, `Semaphore.acquire`, `Condition.await`, and `CountDownLatch.await` on the same synchronizer object in another thread.
- For each pair of threads that successfully exchange objects via an `Exchanger`, actions prior to the `exchange()` in each thread *happen-before* those subsequent to the corresponding `exchange()` in another thread.
- Actions prior to calling `CyclicBarrier.await` and `Phaser.awaitAdvance` (as well as its variants) *happen-before* actions performed by the barrier action, and actions performed by the barrier action *happen-before* actions subsequent to a successful return from the corresponding `await` in other threads.

Demo OutOfOrderTest.java again

BRIAN GOETZ

WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA



JAVA CONCURRENCY IN PRACTICE



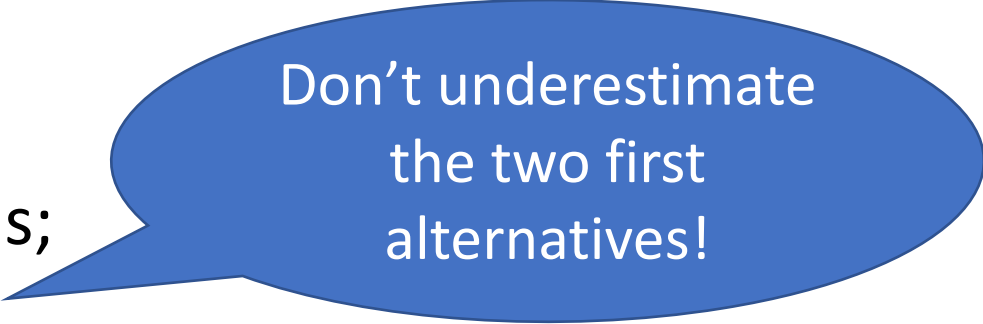
Reading suggestions

- See *Java Concurrency in Practice* (2006) if you want more of this. The book presents simplified rules you can follow to do concurrent programming in Java instead of having to learn the details of the Java memory model.
- E.g., the book provides useful “safe publication idioms”
- Also e.g.: Hans-J. Boehm, “Threads cannot be implemented as a library” (2005).
(<https://doi.org/10.1145/1065010.1065042>)
- Also e.g.: Hans-J. Boehm and Sarita V. Adve, “You don’t know jack about shared variables or memory models” (2012).
(<https://doi.org/10.1145/2076450.2076465>)

Advice from JCP, p. 16

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:

- *Don't share* the state variable across threads;
- Make the state variable *immutable*; or
- Use *synchronization* whenever accessing the state variable.



Don't underestimate
the two first
alternatives!

Summary?

- Make sure to not have data races in your Java programs
- One way to think about all of this: *Atomicity and visibility*
- Visibility aspect new in weak memory models compared to SC!

If you only will remember one thing:

In concurrent programming in Java, not only do we have to consider **atomicity**, we also must consider **visibility**!

visibility

visibility

visibility

visibility

visibility

v i s i b i l i t y