Weak Memory Models

Lecture X of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg

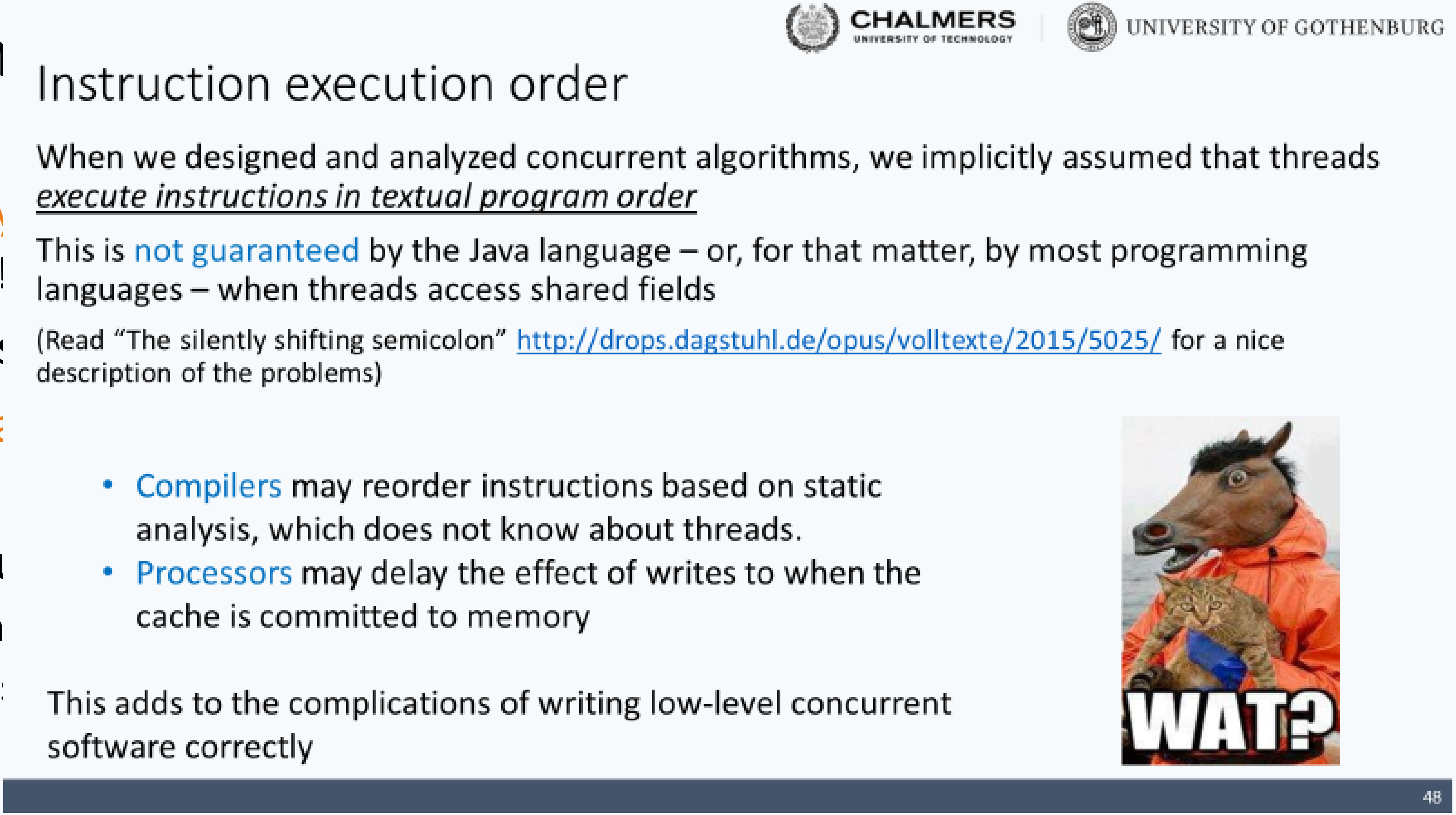
Based on material prepared by Andreas Lööw

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY

Telling th

- Why synch
  - Atomicity
  - Visibility!
- We have us
- Real langua
  - Memory
- In this lectu
  - Rudimen
  - Principle:



Instruction execution order

When we designed and analyzed concurrent algorithms, we implicitly assumed that threads *execute instructions in textual program order*

This is not guaranteed by the Java language – or, for that matter, by most programming languages – when threads access shared fields

(Read “The silently shifting semicolon” http://drops.dagstuhl.de/opus/volltexte/2015/5025/ for a nice description of the problems)

- Compilers may reorder instructions based on static analysis, which does not know about threads.
- Processors may delay the effect of writes to when the cache is committed to memory

This adds to the complications of writing low-level concurrent software correctly

Today’s menu

- 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 JMM

What are memory models?

Memory Models

- As part of language semantics:
  - How threads communicate through shared memory.
  - What values are variable reads allowed to return?
- There are different memory models:
  - Sequential Consistency – one of the "strongest" memory models. Often assumed for pseudocode (and up to now in this course).
  - Java uses Java Memory Model (JMM) – a weak memory model.

Reading variables: Sequential programming

```
int x = 0;
int y = 0;
x = 1;
y = 1;
print(y);
print(x);
```

What value will this read of y return?

Obviously 1! We always get the latest value!

Reading variables: Concurrent programming

```
bool done = false; int res = 0;

green_thread {
  res = 666;
  done = true;
}

blue_thread {
  if (done)
    print(res);
}
```

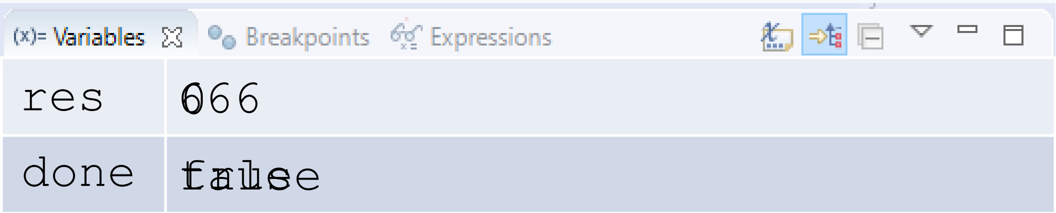
What are the possible outcomes of running?

Let’s consider all possible interleavings.

Reading variables: Concurrent programming

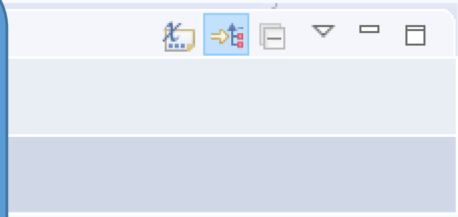
```
bool done = false;
int res = 0;
green_thread {
1◀ res = 666;
2◀ done =
3}◀

blue_thr
1◀ if (done)
2◀   print(res);
3}◀
```

1;1;2; No output

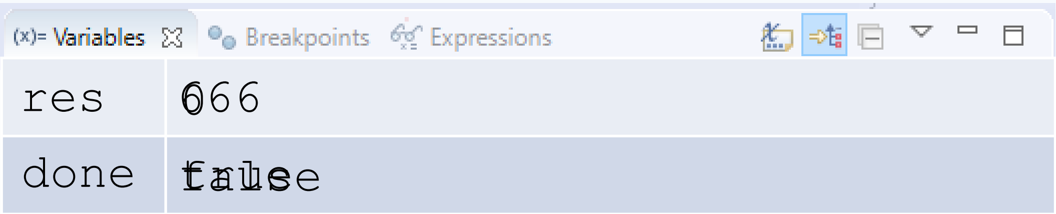


res	666
done	false

1;1;2; No output



1;2;1;2; Output 666



res	666
done	false

Conclusion:

Either output nothing or 666

Let’s see what Java says …

Demo OutOfOrderTest.java

Reading variables: Sequential consistency (SC)

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 all weak memory models (e.g. JMM)!

Interleaving-based semantics is the “obvious” semantics.

Why make things more difficult? Why give up program order?

Because sequential consistency costs too much.

```
bool done = false;
int res = 0;
green_thread {
  res = 666;
  done = true;
}

blue_thread {
  if (done)
    print(res);
}
```

Take home message 1

You must understand the memory model in order to write correct programs.

Why weak-memory models?

SC problem 1: Compiler optimizations

For some compiler optimizations we want to reorder writes to variables.

This does not happen in pseudocode …

Messy details …

SC problem 1: 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:

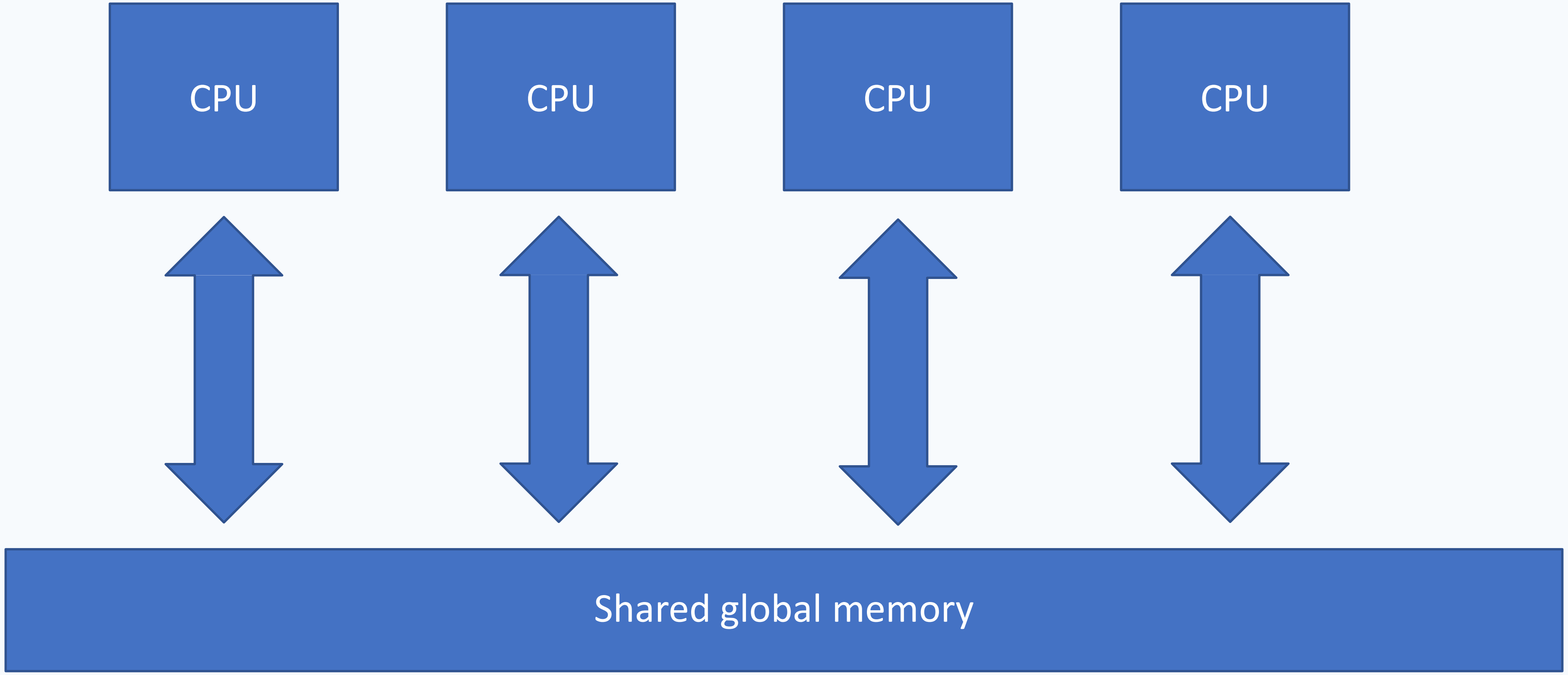
```
y = 2;
x = 1;
z = x + y; // x = 1, y = 2,
z = 3
```

Write order swapped

Write order swapped

SC cost 2: Causes too much cache synchronization

Cost of SC not obvious with too simplified machine models:



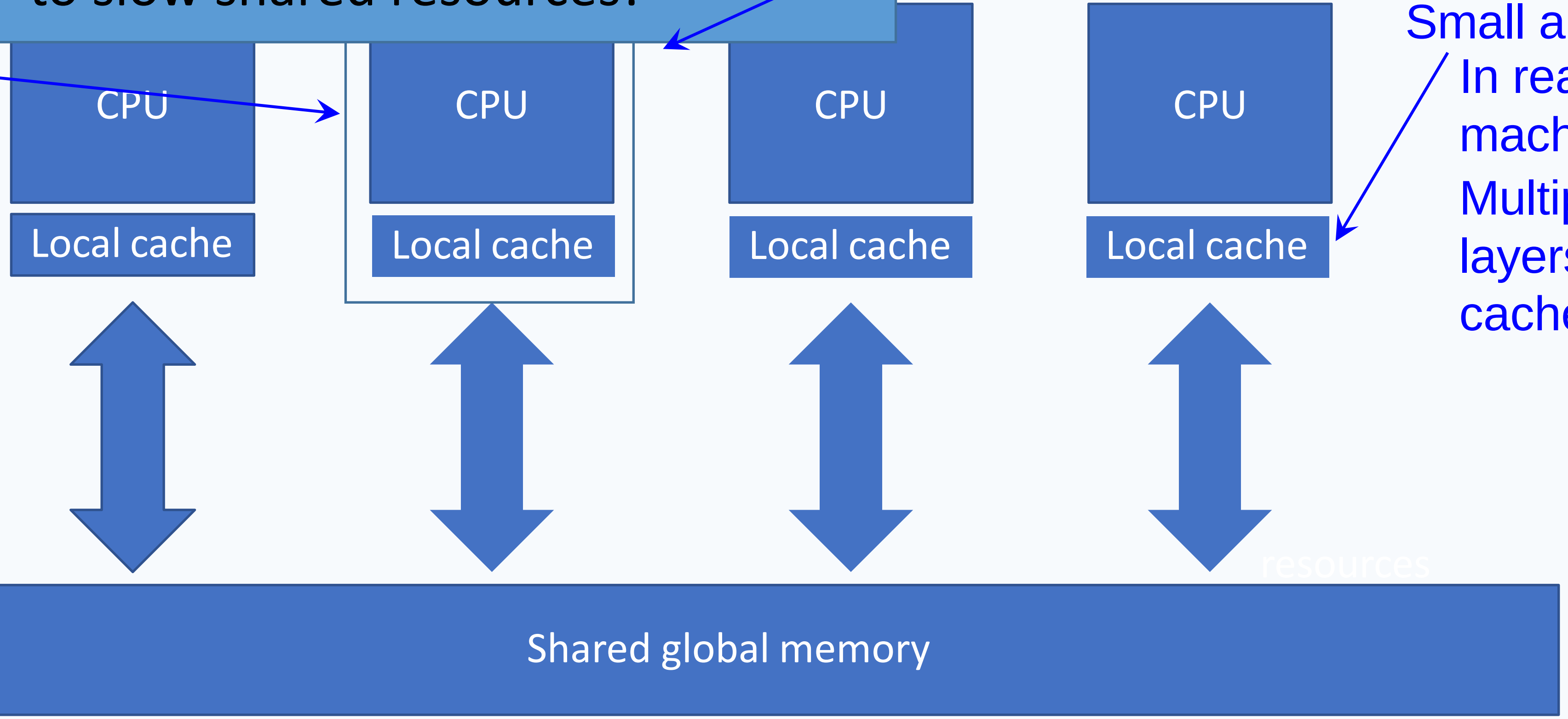
SC cost 2: Causes too much cache

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

Slightly m nputers:

Want to keep computations local (avoid communication overhead)

In modern CPUs, even a single CPU may execute out of order and in parallel …



Small and fast

In real machines: Multiple layers of cache!

Large (but slow) shared memory

Why not SC?

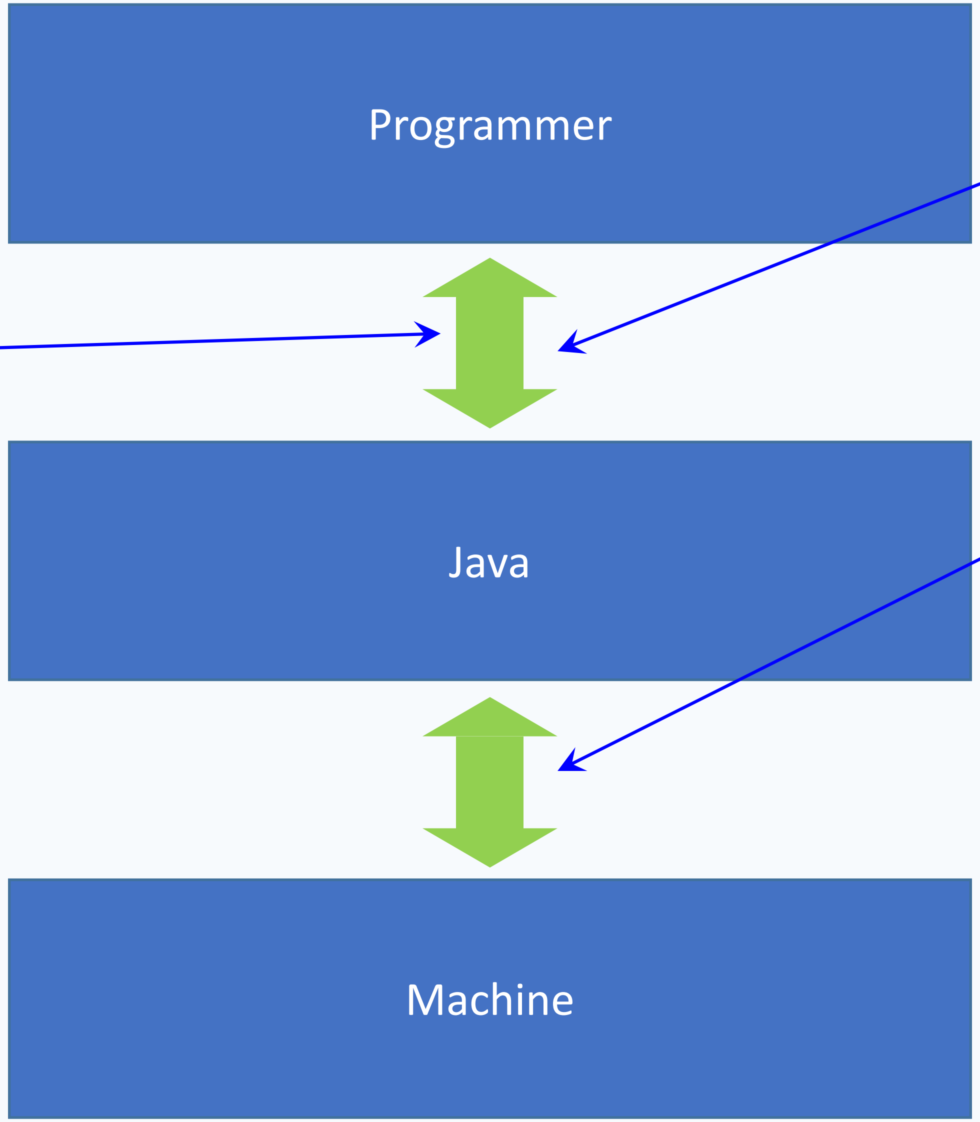
- Examples:
  - Out of order execution
  - Compiler optimizations
  - Avoid communication
- SC too expensive in many situations
- Solution to mentioned problems:
  Relax some guarantees offered by SC → we get weak memory models

  Weaker memory models (potentially) more performant, but more difficult to program in

Something about JMM

Example of a weak memory model

More context: machine details



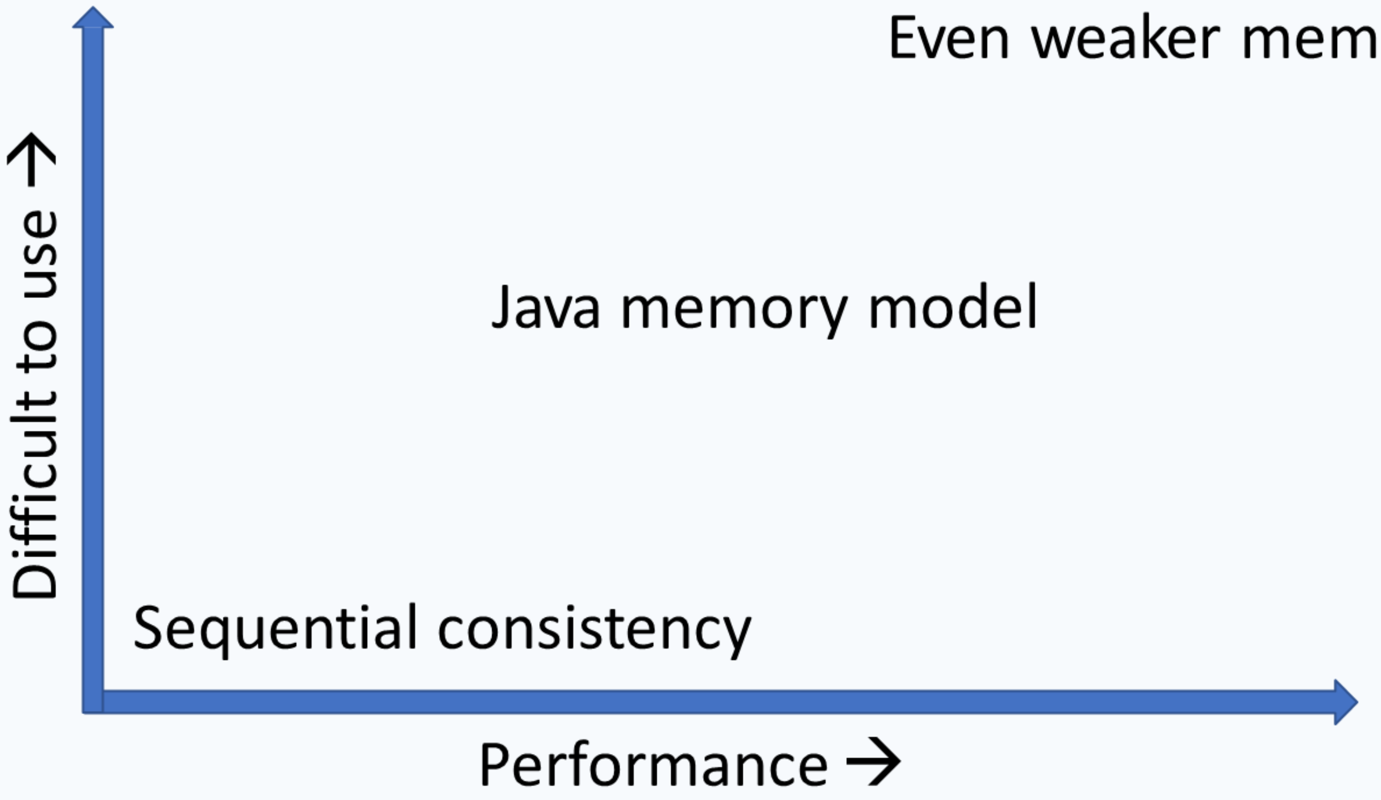
Program in Java memory model

Java protects us from assembly language and from the machine’s memory model.

Java compiler developers implement Java memory model in the memory model of the underlying machine (different machines have different memory models)

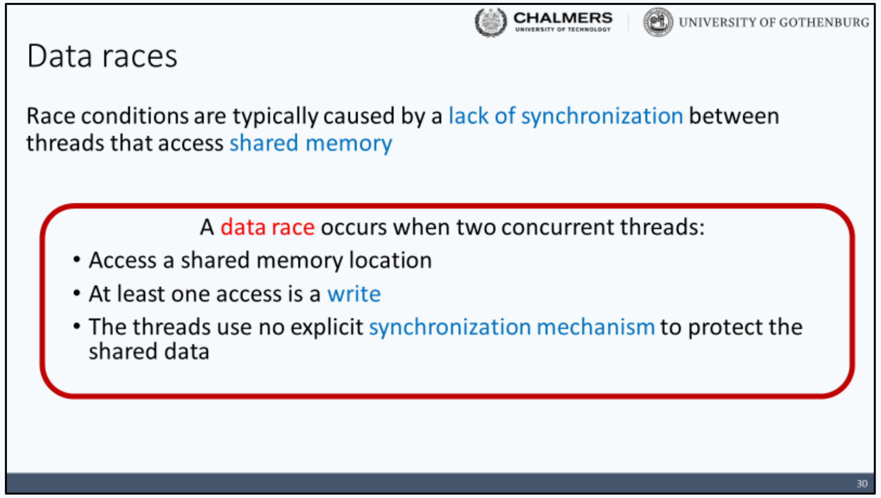
The Java memory model

- Less convenient than SC, but implementable on modern machine architectures without too much performance loss
- There is no “right design”:



SC for data-race-free programs

- A few 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

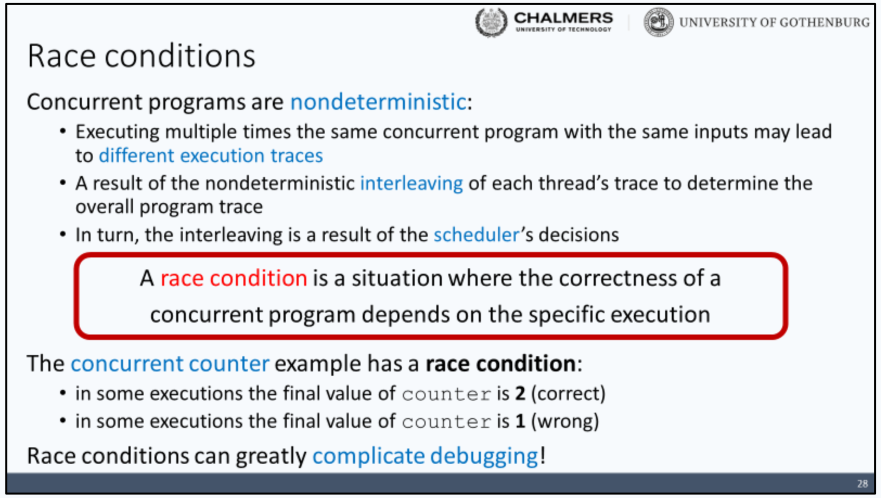


Data races

Race conditions are typically caused by a lack of synchronization between threads that access shared memory

A data race occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a write
- The threads use no explicit synchronization mechanism to protect the shared data



Race conditions

Concurrent programs are nondeterministic:

- Executing multiple times the same concurrent program with the same inputs may lead to different execution traces
- A result of the nondeterministic interleaving of each thread’s trace to determine the overall program trace
- In turn, the interleaving is a result of the scheduler’s decisions

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

The concurrent counter example has a **race condition**:

- in some executions the final value of `counter` is **2** (correct)
- in some executions the final value of `counter` is **1** (wrong)

Race conditions can greatly complicate debugging!

Data races: slight (Java) variation

Def.

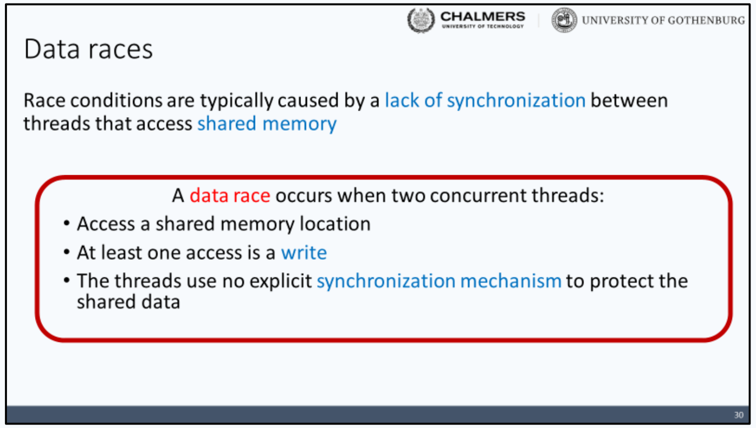
Two memory accesses are in a data race iff they access the same memory location simultaneously (they are interleaved next to each other), at least one access is a write, insufficient explicit synchronization used to protect the accesses

Def.

A program is data-race-free iff no SC execution of the program contains a data race

Notes:

- We quantify over all SC executions in the second
- Data-race-freedom is a “language-level” property!



Data races

Race conditions are typically caused by a lack of synchronization between threads that access shared memory

A data race occurs when two concurrent threads:

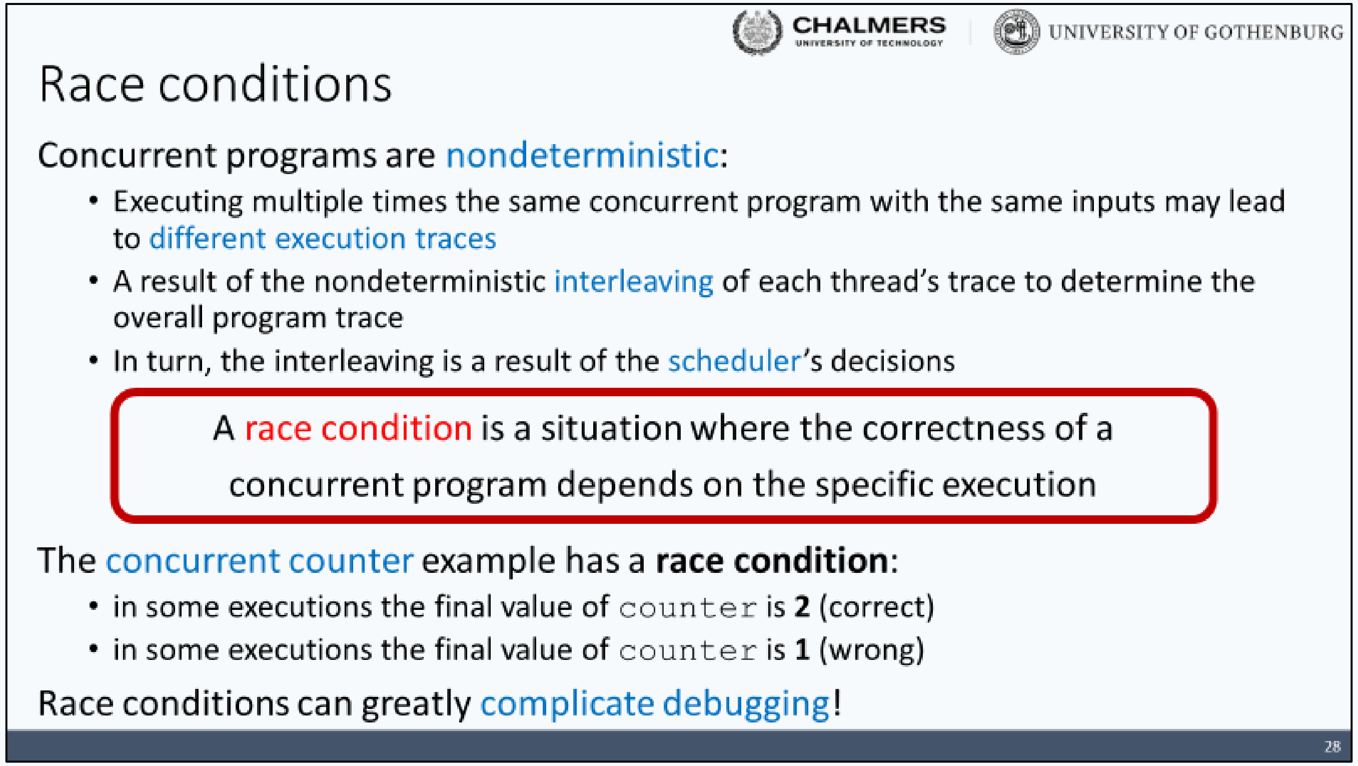
- Access a shared memory location
- At least one access is a write
- The threads use no explicit synchronization mechanism to protect the shared data

Definition of data race surprisingly subtle

Does this program contain any data races?

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

t1 {
 if (x) y = true;
}


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

Race conditions



CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Race conditions

Concurrent programs are nondeterministic:

- Executing multiple times the same concurrent program with the same inputs may lead to different execution traces
- A result of the nondeterministic interleaving of each thread's trace to determine the overall program trace
- In turn, the interleaving is a result of the scheduler's decisions

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

The concurrent counter example has a **race condition**:

- in some executions the final value of `counter` is **2** (correct)
- in some executions the final value of `counter` is **1** (wrong)

Race conditions can greatly complicate debugging!

28

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
- Reasoning principle in more detail:
  1. Assume SC and make sure that there are no data races
  2. If no data races, we can assume SC when reasoning about race conditions
- 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

Programming in the JMM

As an example of a weak memory model

What does all 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 than with SC
  - Sprinkle additional synchronization information on top of your program
  - Variable qualifiers, synchronization mechanisms (e.g. locks), etc.
  - Exactly what “annotate” means depends on language

- Essentially, you annotate which data/actions are shared and which are not

Simpler example: only one variable!

```
bool done = false;


t1 {
  done = true;
}


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

- Does this program contain
  - data races?
  - ditions?

es, done is accessed without
on and one of the accesses is a

on = depends on the specification we
(what it means for the program to be

on = even if we had a specification, we
race so our reasoning principle does

- There is a problem with this program!
- From SC perspective, everything is fine!
- No atomicity problems … but visibility problems!

Simple example (fixed)

```
volatile done = false;


t1 {
  done = true;
}


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

- Solution: Annotate your program. E.g., in Java `volatile` is considered synchronization.
- Does this program contain
  - data races?
  - race conditions?
- Data race = no, in Java volatileaccesses are considered synchronized
- Race condition = still depends on specification
- Example spec: “If the program outputs something, it must output 33”.
- Race condition = no, for the above specification the correct output does not depend on specific execution/interleaving.
- Example spec: “The program outputs 33”.
- Race condition = yes, some interleavings give us the 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?

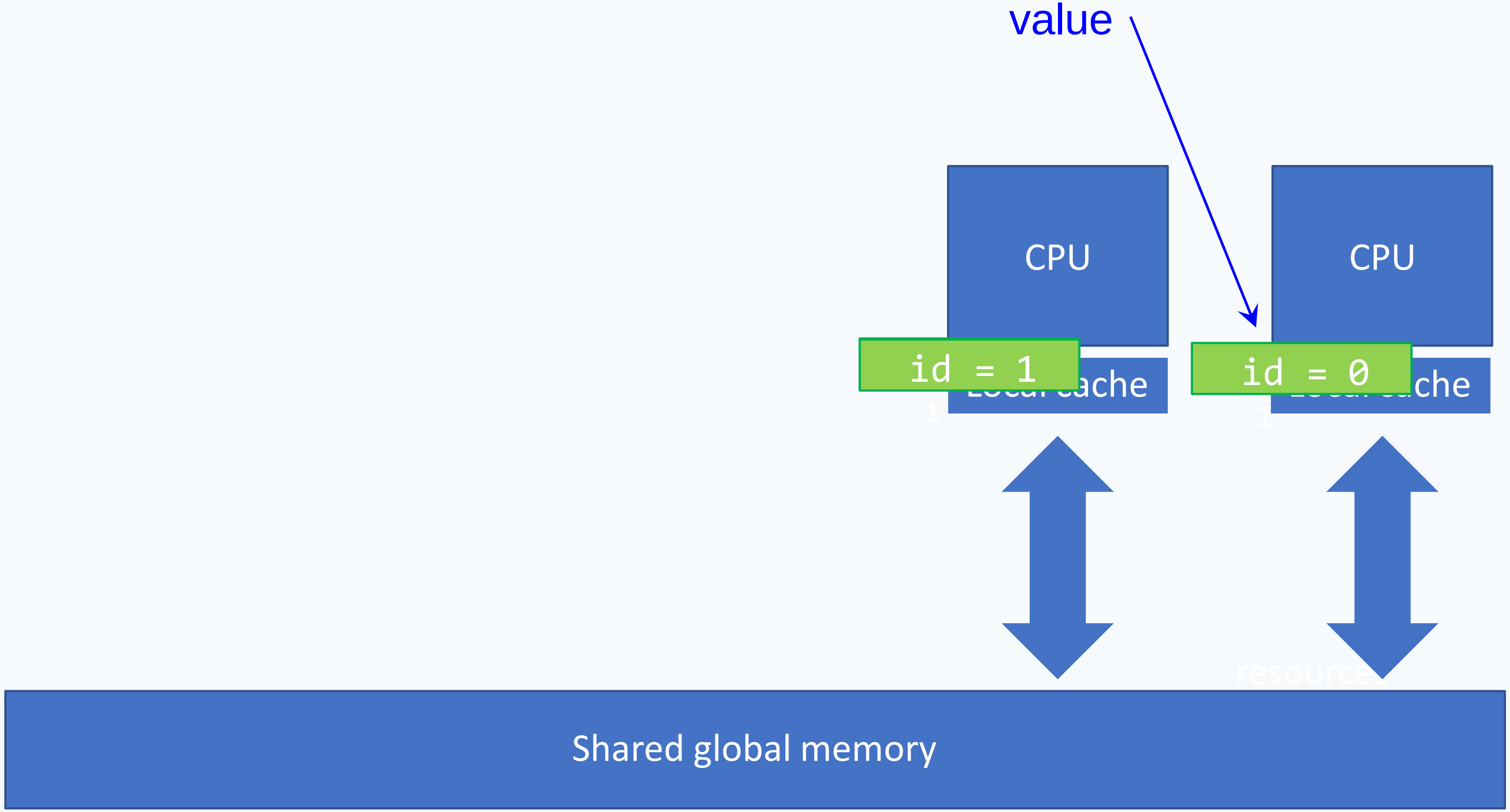
We have a race! All accesses to the shared variable done must be synchronized!

Here we have (again) atomicity, but not visibility

id might exist as multiple copies...

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


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


t2 {
  print(id);
}
```

Might read “stale” value



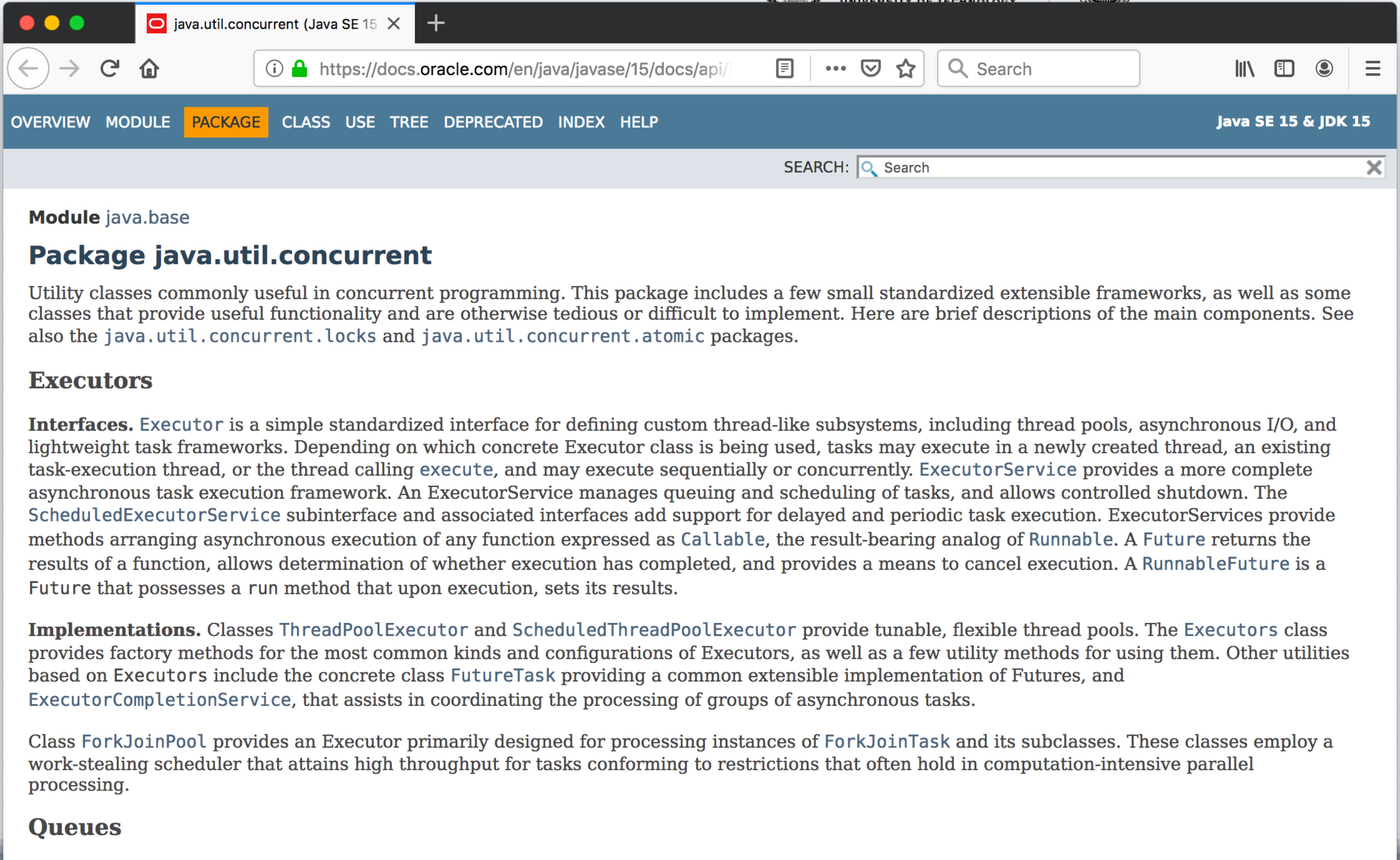
Similar example, with locks (fixed)

```
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

Now there are no data races.

JMM in More Detail



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

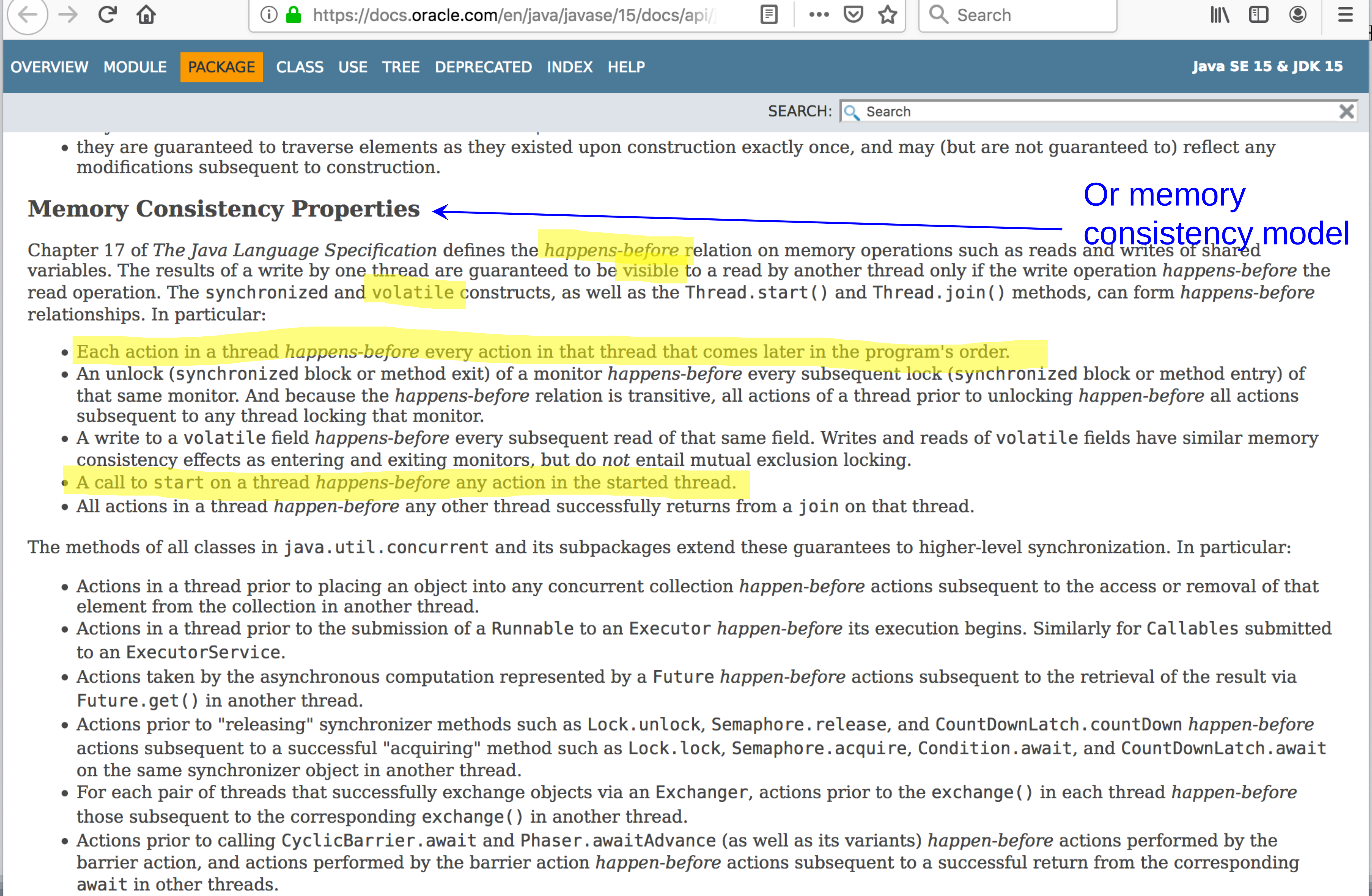
Interfaces. `Executor` is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous I/O, and lightweight task frameworks. Depending on which concrete Executor class is being used, tasks may execute in a newly created thread, an existing task-execution 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

IENBURG



- 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

Or memory consistency model

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 `Callables` 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.

Data races defined in terms of happens-before

From the Java language specification (v. 15):

Two accesses to (reads of or writes to) the same variable are said to be conflicting if at least one of the accesses is a write.

[…]

When a program contains two conflicting accesses (§17.4.1) that are not ordered by a happens-before relationship, it is said to contain a data race.

[…]

A program is correctly synchronized if and only if all sequentially consistent executions are free of data races.

[…]

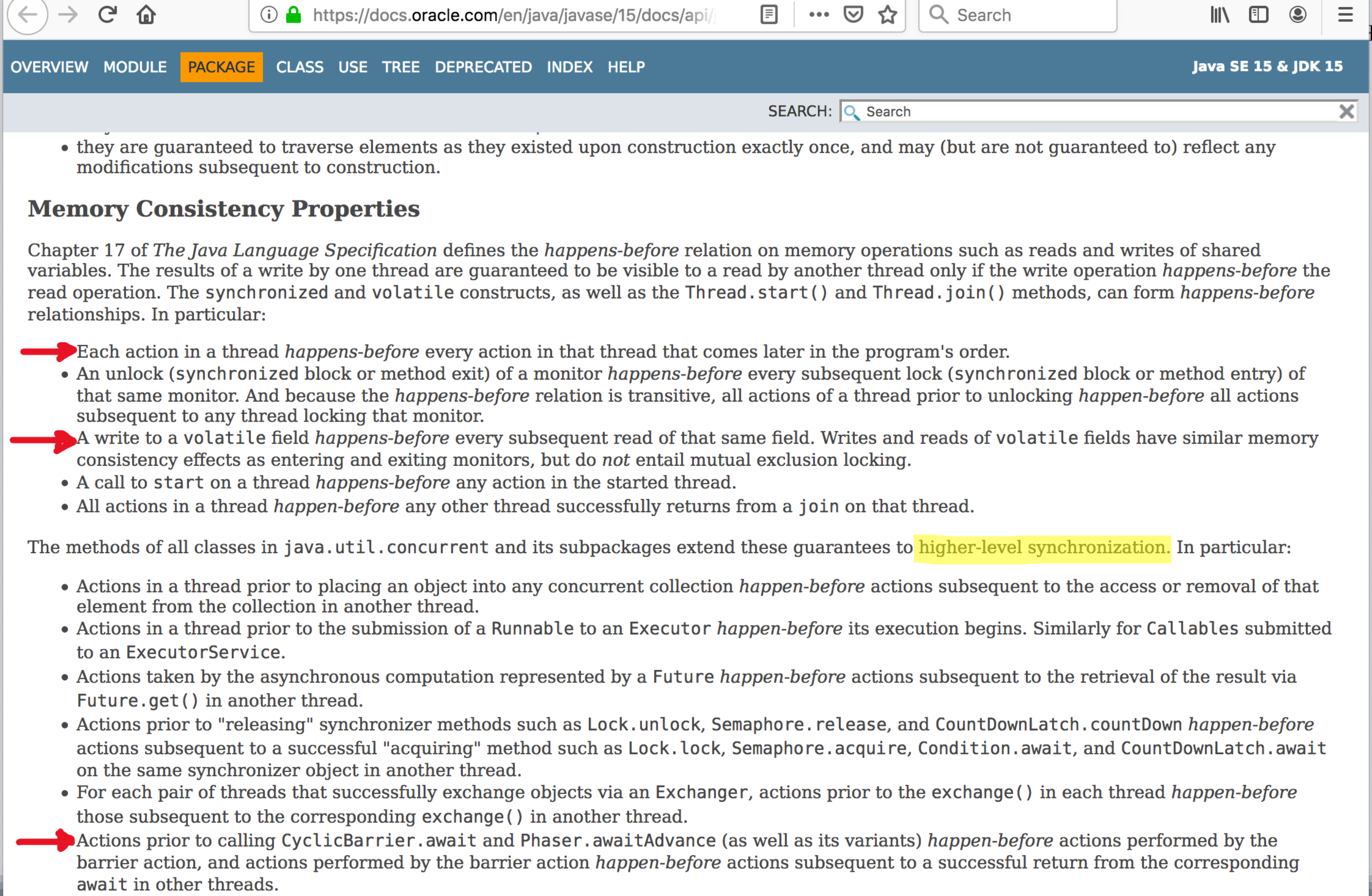
If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent (§17.4.3).

Happens-before example

```
static int x = 1;
x = 2;
Thread t = new Thread(() ->
          System.out.println(x));
x = 3;
t.start();
```

- Data race because t reads x without synchronization?
- (Could argue read and write not overlapping in any SC execution.)
- x write *happens-before x read,* because *happens-before* transitive

IENBURG



- 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 `Callables` 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

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!