

# Concurrency, the Java Memory Model and You

# Preface

- What's in here:
  - Basics of how the JVM handles memory access in the context of concurrency
- What's not (yet) in here:
  - All things Garbage Collector
  - Details on JSR-166 (`java.util.concurrent`)

# TOC

- Why a memory model?
- Theory
- Practice
- What does this mean for the working programmer?

# Why a memory model?

- Provides rules for access to the main memory by threads
- Provides guarantees to the programmer about runtime behaviour
- Rules about memory access can be reasoned about to debug or validate concurrent code.

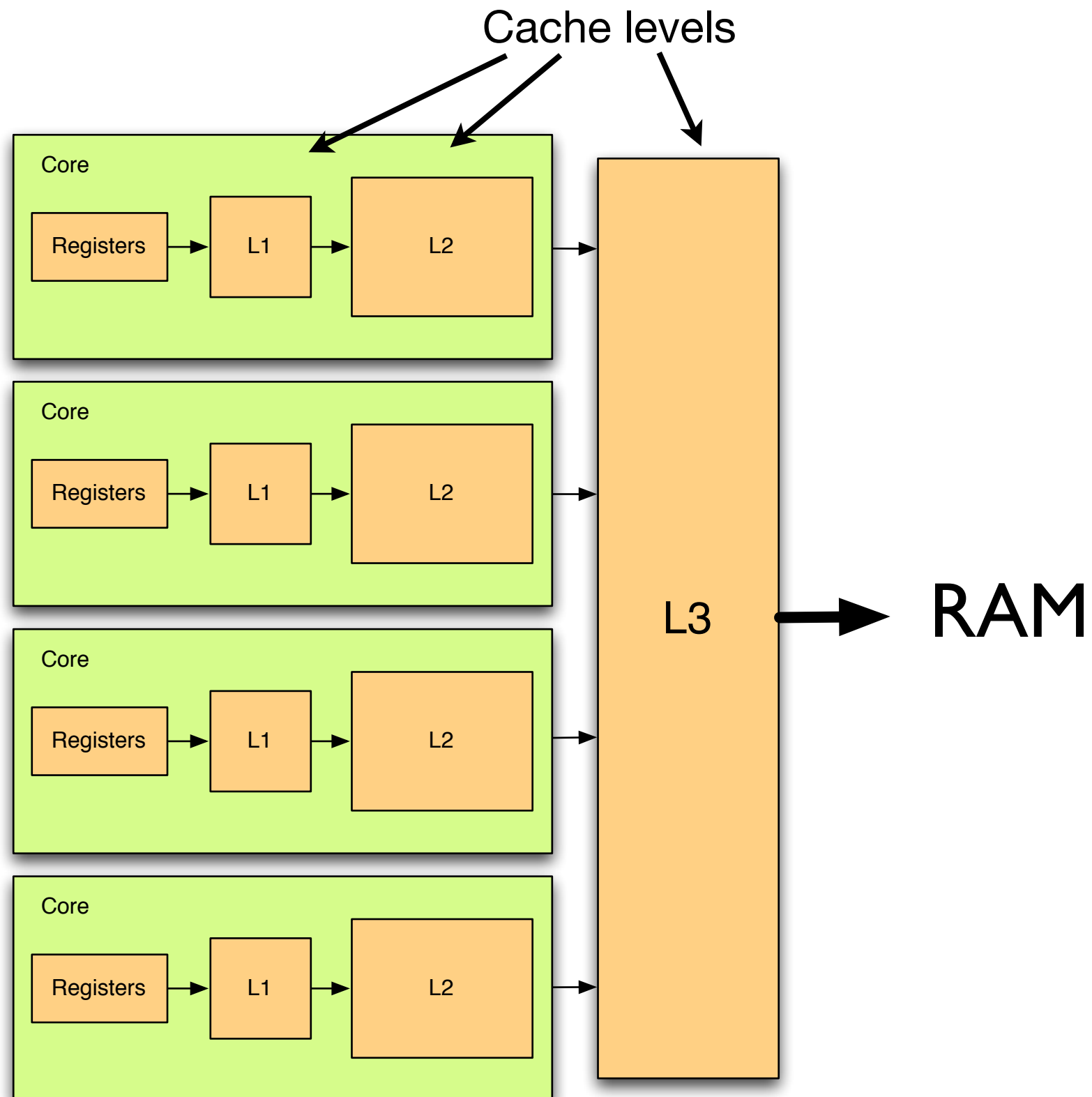
# The Basics

- Working memory per thread
  - thread-confined variables not visible from the outside
  - need to synchronize shared variables between shared and thread memory

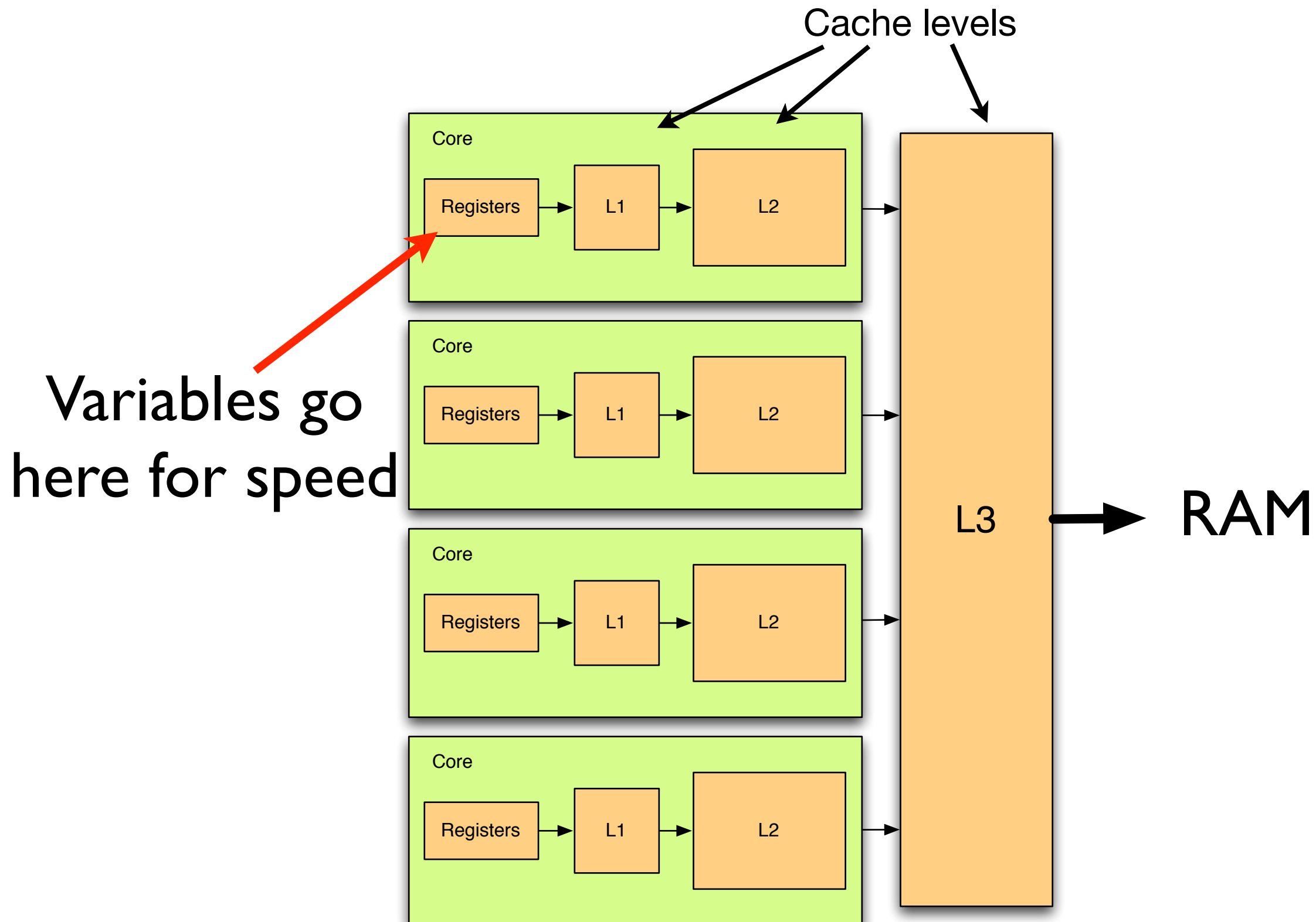
# The Basics

- If synchronization is not made clear to the compiler, you end up with surprising results.
- Compiler optimizations may change code in several ways:
  - Statement Reordering
  - Forward substitution of variables
- CPU might further change things
  - Out-of-order execution
  - Microcode optimizations

# Why does this make sense?

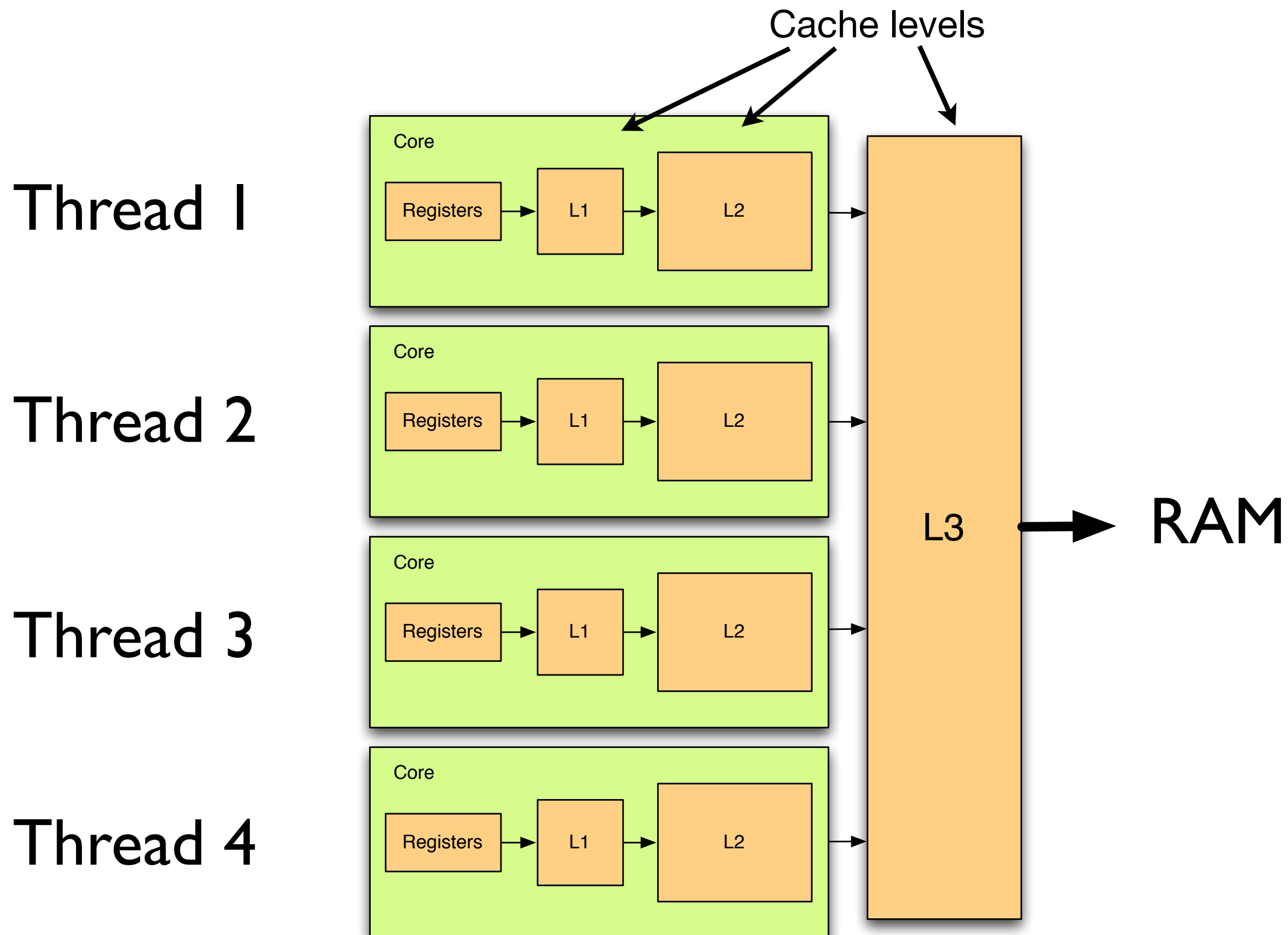


# Why does this make sense?





# Why does this make sense?



# Why does this make sense?

- Variables stay in Registers longer
  - minimizes memory access
- Need for a write back before accessing variable from another core
  - Memory fences

# How does Java model this?

- All actions in a program are governed by a partial ordering.
- If the `happens_before` relation from action A to B is valid, A is guaranteed to happen before B.
- If two write accesses to a variable don't have a `happens_before` relationship you have a data race.

# Actions

- Non-volatile Read/Write
- Synchronized:
  - Volatile Read/Write
  - Locking/Unlocking a monitor
  - Create/Join of a thread
  - External actions
  - Thread divergence (infinite loop)

# happens\_before ordering

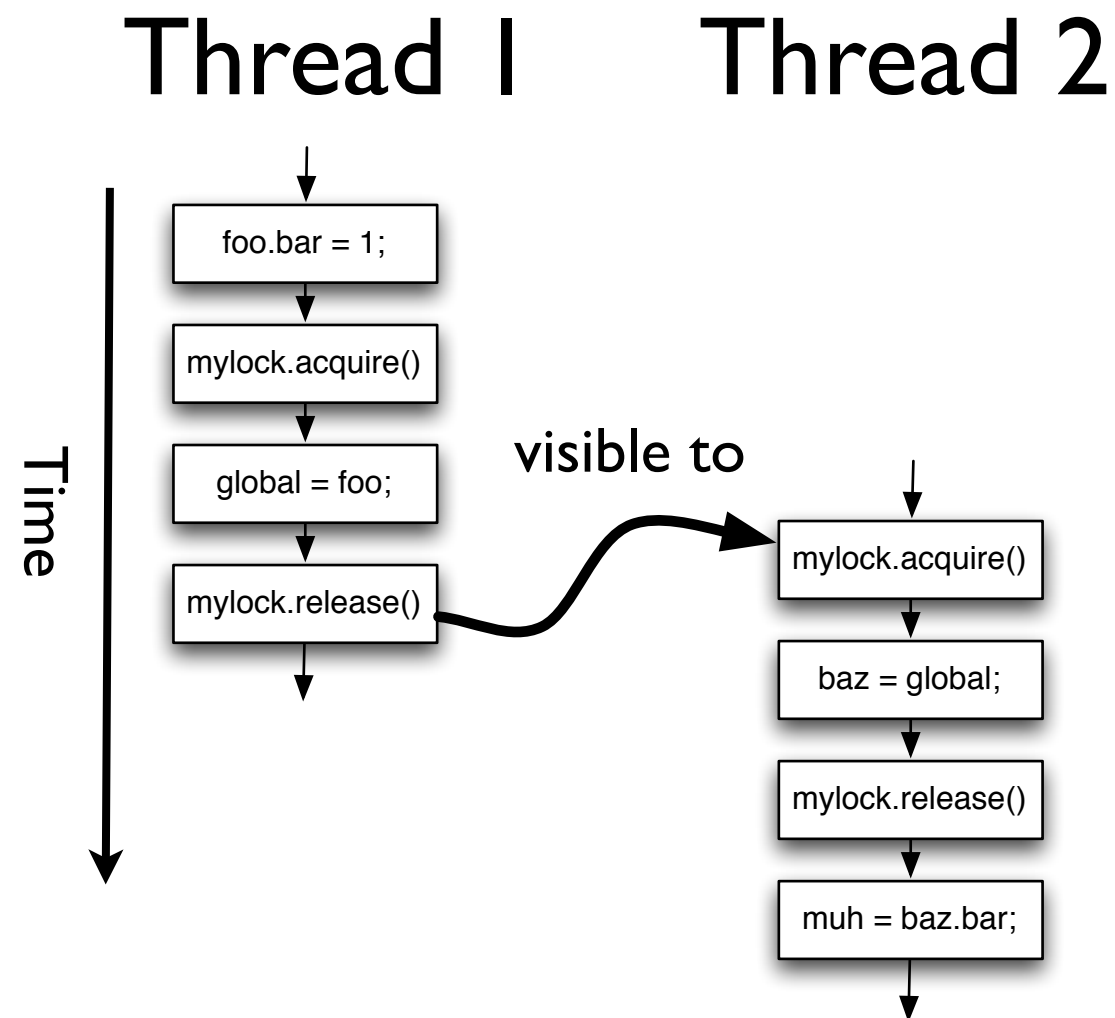
- Let  $x, y$  be actions.
  - If  $x$  and  $y$  are in the same thread and  $x$  comes before  $y$  then  $x$  happens\_before  $y$ .
  - The end of construction for an object happens\_before the start of the its finalizer.
  - An unlock of a monitor happens\_before any subsequent lock on that monitor.
  - A write to a volatile field happens\_before any subsequent read from that field.
  - A call to a threads start() method happens\_before any actions inside the started thread.
  - The last action in a thread happens\_before any other thread successfully returns from a call to join() on that thread.

# happens\_before ordering

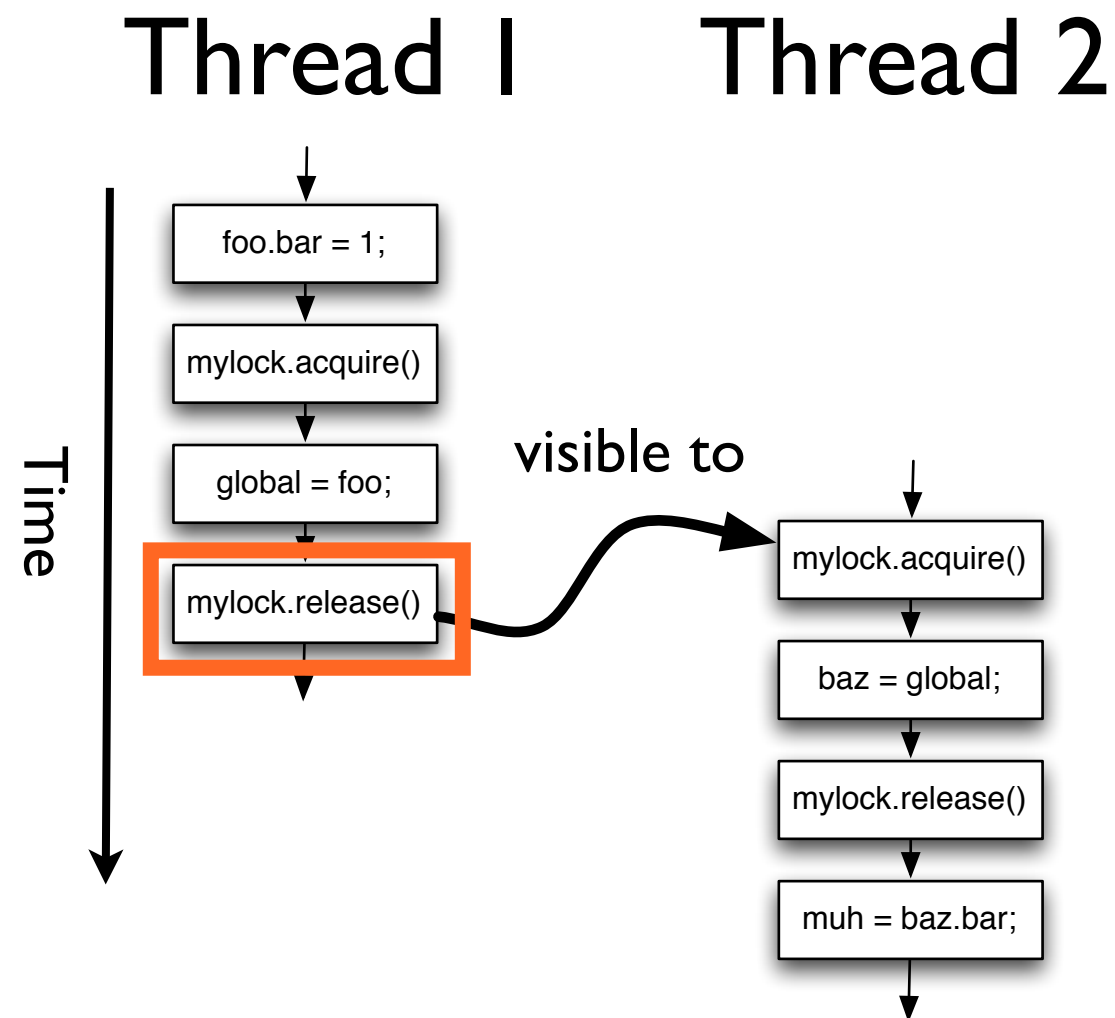
If this ordering holds over all conflicting accesses to a variable in a program, that program is said to be `correctly synchronized`.

It is a necessary, but not sufficient constraint on program consistency!

# Visibility between threads



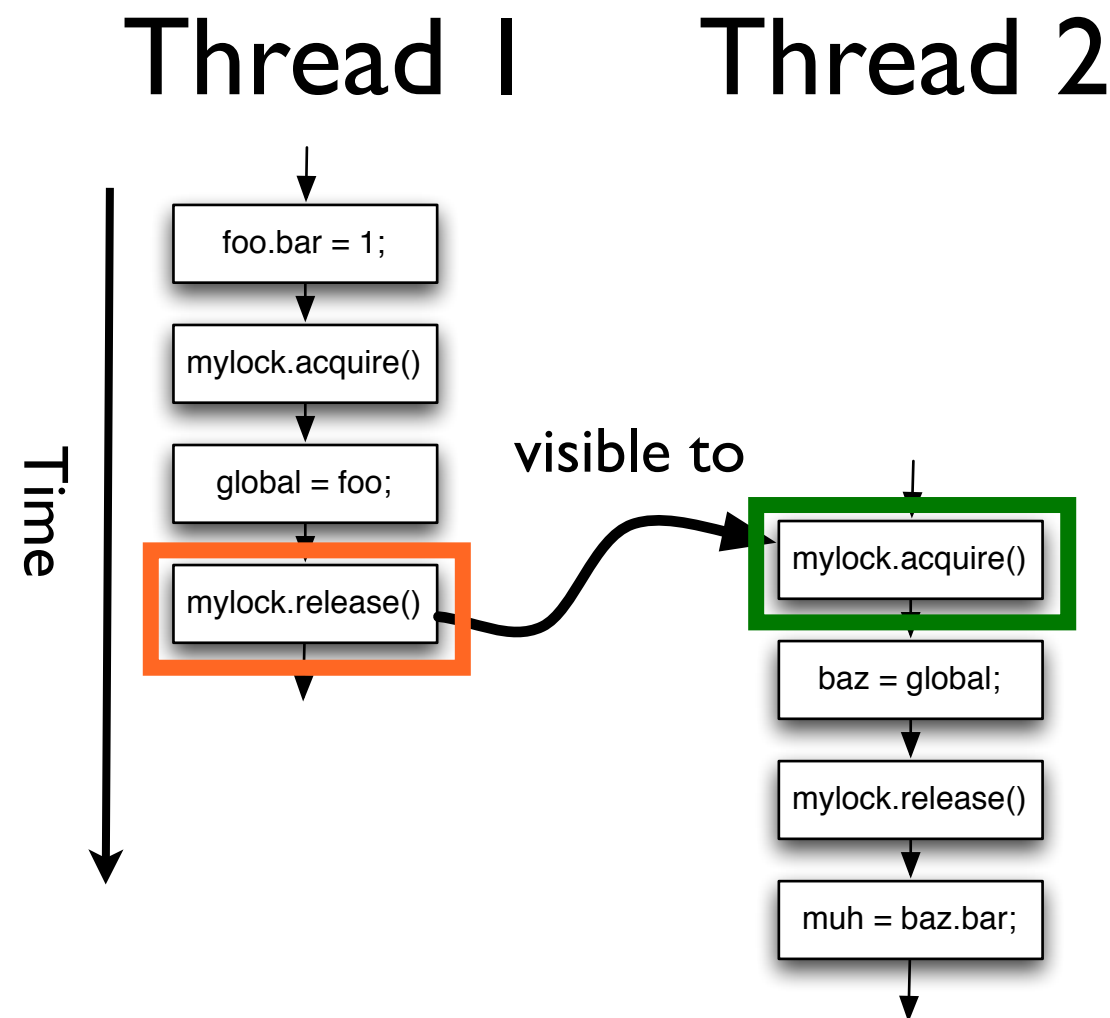
# Visibility between threads



Everything before **this** unlock



# Visibility between threads



Everything before **this** unlock  
is visible to everything after **this** lock

# Example

$A == B == 0$

Thread 1	Thread 2
$B = 1;$	$A = 2;$
$r2 = A;$	$r1 = B;$

`happens_before` consistent outcomes:

```
B = 1;  
A = 2;  
r2 = A;  
r1 = B;
```

```
r2 = A;  
r1 = B;  
B = 1;  
A = 2;
```

# What does this mean in practice?

```
import java.util.concurrent.TimeUnit;

public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested) {
                    i++;
                }
                System.out.println(i);
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

# What does this mean in practice?

```
import java.util.concurrent.TimeUnit;

public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested) {
                    i++;
                }
                System.out.println(i);
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

**UNDEFINED BEHAVIOUR!**

# Corrected program

```
import java.util.concurrent.TimeUnit;

public class StopThread {

    private static volatile boolean stopRequested;

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested) {
                    i++;
                }
                System.out.println(i);
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

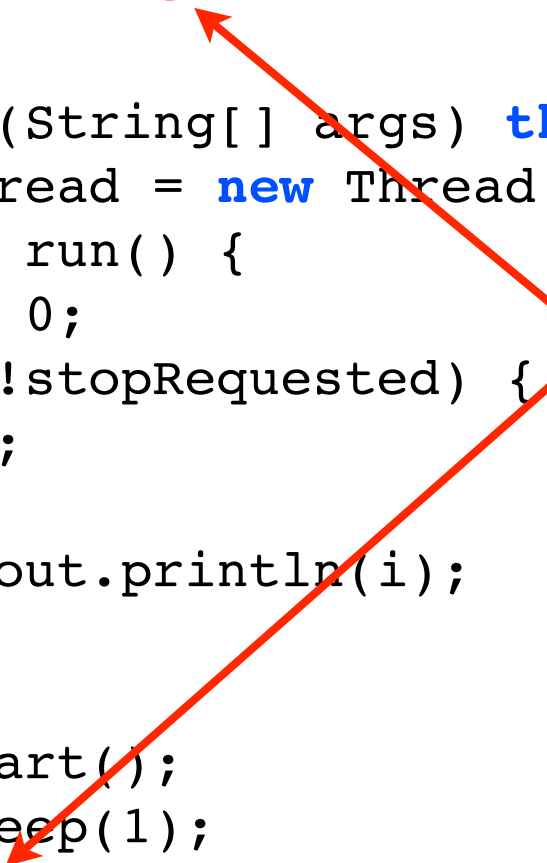
# Corrected program

```
import java.util.concurrent.TimeUnit;

public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested) {
                    i++;
                }
                System.out.println(i);
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

**happens\_before  
must remain valid!**



# A note on volatile

- volatile variables are always directly written to memory.
- access to volatile longs and doubles is atomic (access to non-volatile versions is NOT!)

# The semantics of `final`

- Guarantee to the compiler about access to the `final` field
- No writes means maximum reordering flexibility.
- No synchronization necessary when dealing with immutable objects



# Safe publication

- An object is only in a consistent state once properly initialized.
- Publishing a reference to an object before construction is finished means doom!  
DOOM!

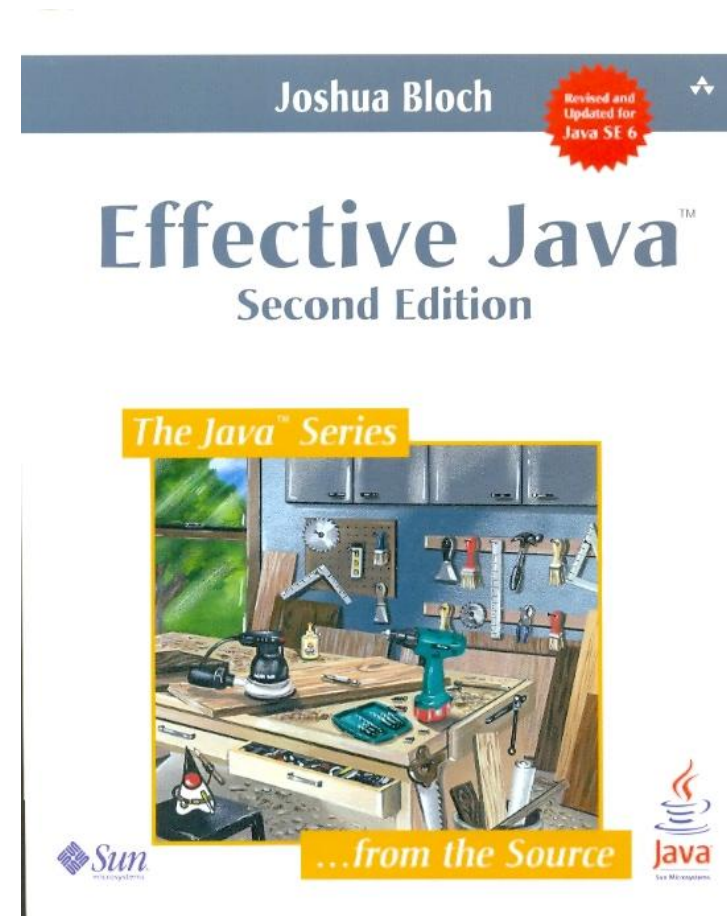
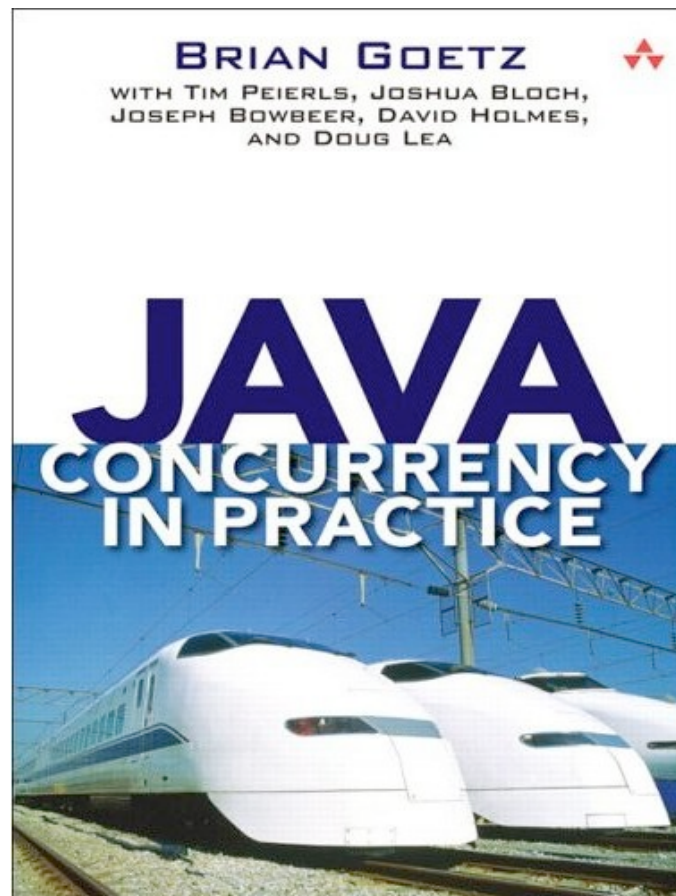
# Safe publication

```
public class Holder {  
    private int n;  
  
    public Holder(int n) {this.n = n;}  
  
    public void assertSanity() {  
        if (n != n)  
            throw new AssertionError("WTF");  
    }  
}
```

# Safe Publication idioms

- To safely publish an object, proper construction must be assured.
- Store it in a `volatile` field or `AtomicReference`
- Store it in a `final` field
- Guard the publication using a lock
- Use a static initialization block

# Primary Sources



Java 7 SE Language Specification: <http://docs.oracle.com/javase/specs/jls/se7/html/index.html>

Java 7 Virtual Machine Specification: <http://docs.oracle.com/javase/specs/jvms/se7/html/index.html>

Java 5 Virtual Machine Specification: <http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html>

William Pugh's page on the Java Memory Model: <http://www.cs.umd.edu/~pugh/java/memoryModel/>