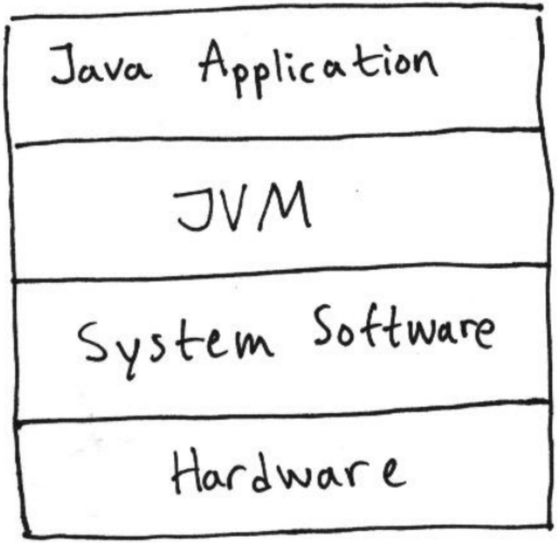
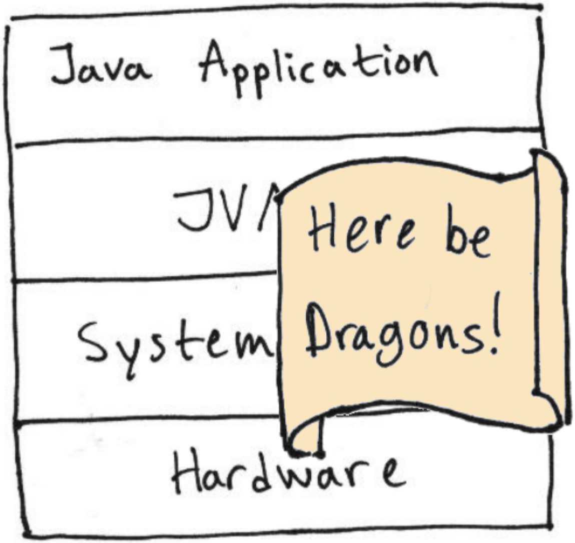
The Java Memory Model

Jesper Öqvist

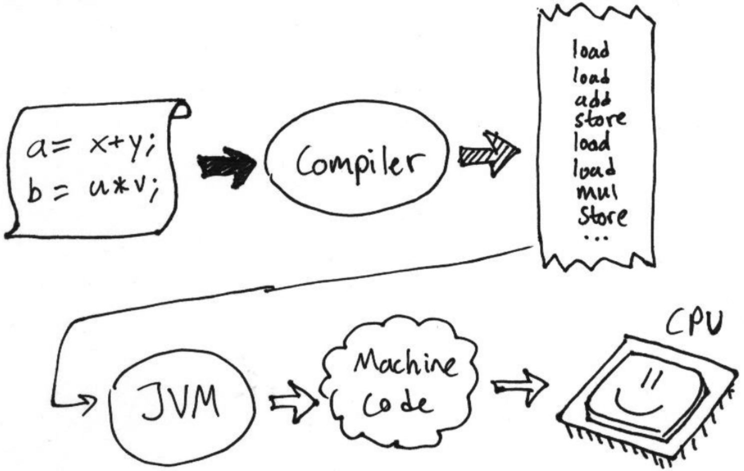
The Java Environment



The Java Environment



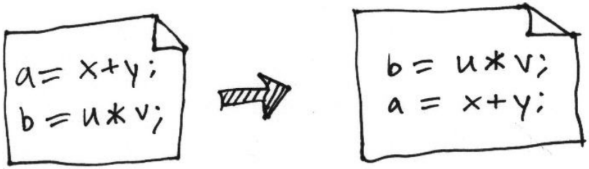
Java Execution



Possible reordering due to Compiler, JVM, or CPU! Plus visibility problems due to CPU caches!

Reordering

Q: Why are operations reordered?
A: To increase performance!

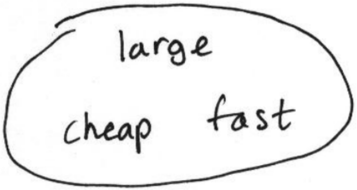


Doing the add after the mul can avoid a couple stall cycles in the CPU, depending on what follows.

Caches

Q: What about the visibility? Why do we need caches?

Computer memory:



A: Choose two!

Caches

- Main memory is large and cheap.
- How slow? Typically 100-300 clock cycles slow.
- We really really want to avoid main memory, but how?

- Store commonly used stuff in smaller, cheaper memory called caches!
- An L1 read takes only about 3-5 cycles!

Caches

- Main memory is large and cheap.
- How slow? Typically 100-300 clock cycles slow.
- We really really want to avoid main memory, but how?

- Store commonly used stuff in smaller, cheaper memory called caches!
- An L1 read takes only about 3-5 cycles!

Caches

- Main memory is large and cheap.
- How slow? Typically 100-300 clock cycles slow.
- We really really want to avoid main memory, but how?

- Store commonly used stuff in smaller, cheaper memory called caches!
- An L1 read takes only about 3-5 cycles!

Caches, cont.

Each core has it's own cache, so now we have cache coherence problems!
These nitty gritty details we leave to the hardware and JVM!

The Java Memory Model

To avoid the headache of knowing what is happening beneath our application, the JMM provides simple rules.

Most importantly: what happens in a single thread is visible in sequential order!

- **Synchronization** -- unlock of an intrinsic lock happens before the locking of the same lock.
- **Volatile fields** -- writing to a volatile field always happens before reading from it.

Publication

Q: Is this safe publication?

```
class Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

A: It depends on how objects of the class are used!

LUND UNIVERSITY

Publication

Q: Is this safe publication?

```
class Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

A: It depends on how objects of the class are used!

```
class A {
    public static Point p = null;

    public void init() {
        p = new Point(2, 3);
    }

    public int sum() {
        if (p == null) {
            init();
        }
        return p.x + p.y;
    }
}
```

It is possible that one thread at some point sees the value 3 of sum, meaning that only the y value was initialized, even though x should have been initialized first!

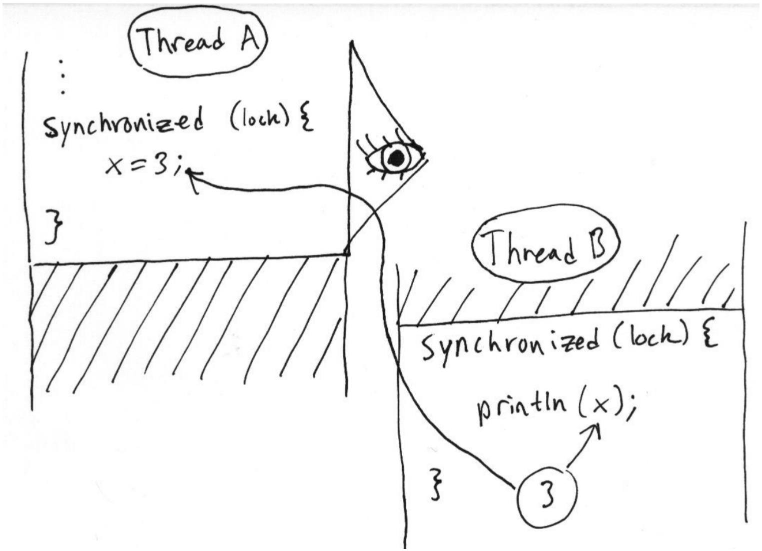
Publication

Obviously the previous example had other problems too.

There was no synchronization around the null check and call to init, allowing several threads to attempt to initialize the point simultaneously.

Synchronization often solves visibility problems.

Had we used synchronization in the Point example then all threads acquiring the intrinsic lock (Point.class could act as lock) would have seen the fully initialized point because the locking happened before the unlocking which happened before the initialization.


Thread A
synchronized (lock) {
x = 3;
}
Thread B
synchronized (lock) {
println(x);
}
3

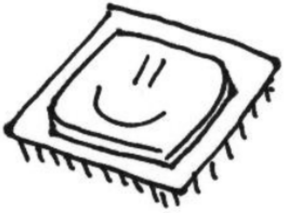
Initialization Safety

- Final fields have special initialization privileges.
- Their initialized value is always visible regardless of how their enclosing objects are published. *

* **IF** the constructor does not leak references to them.

Questions?

Questions?



Exercises

1. Create a class with a static lazily initialized field that is safely published. The field is accessed through a method which can initialize it.
2. Is there any other way to safely initialize the static field? Possibly without the initialization method?