Abstractions for Concurrency

Using raw locks is risky.

- deadlock
- forget to release the lock

```
try {
  l.lock();
  ...
} finally {
  l.unlock();
}
```
Abstractions in the language:

1. Synchronized method -
   ```java
   public synchronized void foo(int param) {
   }
   ```
   on entry, acquire a lock on x when x.foo(3);
   when the method exits or throws, lock is released.

   Adv: Can't forget to release lock!
   Disadv: Harder to get multiple locks

2. Synchronized block
   ```java
   synchronized (x) {
   }  hold lock on x in here
   ```
Threads, Locks & Memory

Thread \( T_1 \)

\[ V = 0 \text{ init.} \]

\[ x = V ; \]

\[ V = 1 ; \]

main mem \( V ? \)

Thread \( T_2 \)

\[ x = V ; \]

\[ V = 2 ; \]

Working Memory (register context)

Load \( V \)

Store \( V \)

Use \( V \)

Assign \( V \)

With locks:

* Getting a lock invalidates working memory
* Releasing a lock flushes working memory

\[ \begin{array}{c|ccc}
\text{vote} & 0 & 1 & 2 \\
\hline
3 & 1 & 2 & 8 \\
\end{array} \]
Most vars are 32 bits (mem. refs., ...)
read/write of a 32-bit var (on a 32-bit processor)
ar atomic

double & long are 64 bits ⇒
treats as 2 variables in hardware
To avoid this:

If multiple threads share a double or a long
and they don't use locks

public volatile long x = ...;

public volatile double y = 3.5;
A better way: ADTs for concurrency

**Example:** BlockingQueue

- producer thread
  - work is enqueued
  - more efficient

- blocking for notification — wait until notified
  - when consumer is getting behind
    - "wait a while"
  - polling — periodically check if there's space in the queue

- consumer thread
  - work is dequeued
  - when there's no work to do
    - "wait a while"
  - polling — periodically check for more work
Synchronized
enqueue(x) {
    inside wait
    until
    space in
    buffer
}

synchronized
degqueue() {
    wait until
    not empty 
    then
    return
    front item
}