Minute essay comments

- Simplicity of multithreading in Scala.
- Inclusion vs. parametric polymorphism? Is-a relationship.
- Saves time or loses time?
Motivation

- The future is parallel.
- Core counts are growing but clock speed isn't and neither is single thread performance.
- Software developers are behind the curve on this.
You can use the java.lang.Thread class to represent a thread.

Pass it a new Runnable that you define a run method in and call start to make it go.

This makes it very easy to start new threads, but there are significant pitfalls when mutable memory is involved.
Synchronization

- Threads use shared memory and you don't get significant control over what happens when.
- Race conditions are errors that occur because of dependence on timing details.
- Bank example.
- You can synchronize on objects to make sure critical blocks aren't accessed in parallel
  - `obj.synchronized { ... }`
- Slow and can cause deadlock.
Allows synchronization between threads. A thread can wait and it won't restart until another thread notifies it.

- Put wait in while loop that checks boolean.
- Always use notifyAll instead of notify. Failure to do so leads to deadlocks.
Java 5 added the java.util.concurrent package and others below it.

Provides better ways to do common tasks for parallel.
Executors

- Use the proper one of these to start threads instead of making them manually.
- Allows Callable[A] and Future[A] which return a value.
Parallel Data Structures

- BlockingQueue
- ConcurrentHashMap
- CountDownLatch
- CyclicBarrier
- Exchanger
- PriorityBlockingQueue
- Semaphore
- Scala provides some support for basic collections.
Locks

- More flexible than synchronized.
- Provides extra power when needed. Particularly for locking across method calls.
Data values with atomic access.
Faster and easier than doing your own synchronization.
How many cores does your computer have? Have you ever tried to keep them all busy?

The next IcP is Friday.