Java Concurrency Idioms

Alex Miller

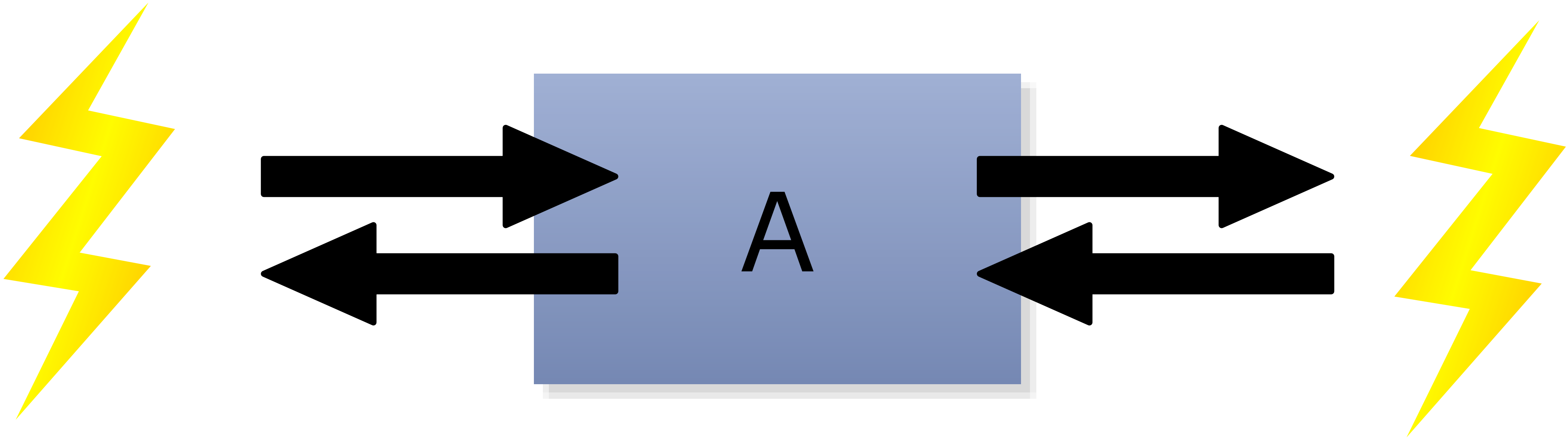
Sharing

Signals

Work

Sharing

Data Race!



Unsafe access

NOT safe for multi-threaded access:

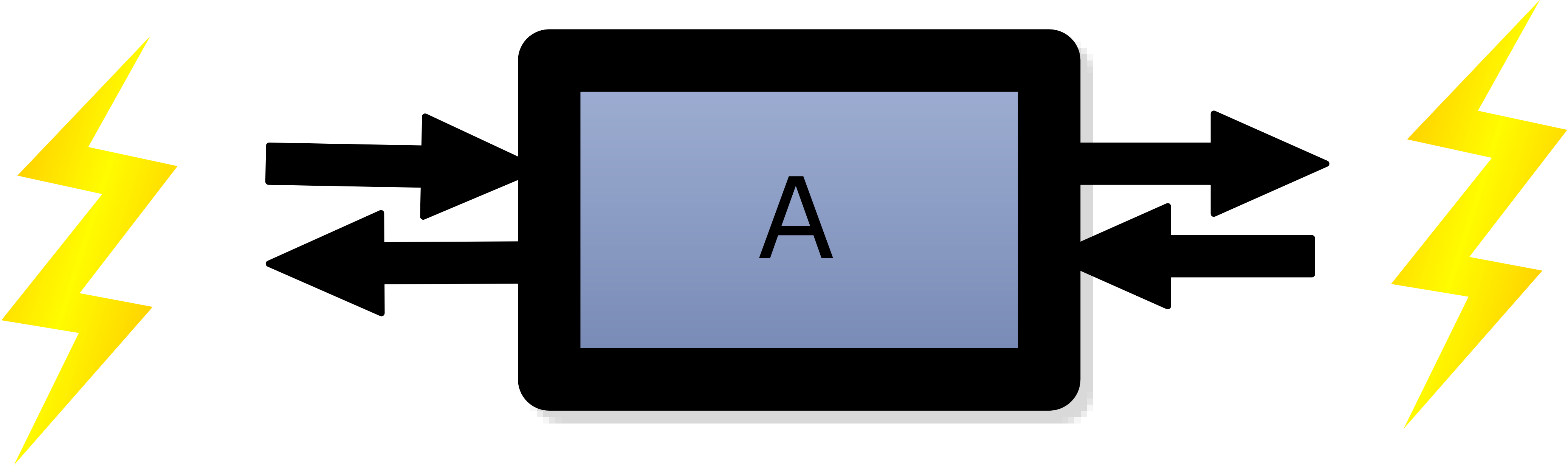
```
public interface Counter
{
    int increment();
}


public class UnsafeCounter implements Counter {
    private int c = 0;

    public int increment() {
        return c++;
    }
}
```

volatile

Is this safe?

```
public class VolatileCounter implements Counter
{
    private volatile int c = 0;

    public int increment() {
        return c++;
    }
}
```

Synchronization



synchronized

```
public class SynchronizedCounter
   implements Counter {

    private int c = 0;

    public synchronized int increment() {
        return c++;
    }
}
```

Atomic classes

```
public class AtomicCounter implements Counter
{
    private final AtomicInteger c =
                         new AtomicInteger(0);

    public int increment() {
        return c.incrementAndGet();
    }
}
```

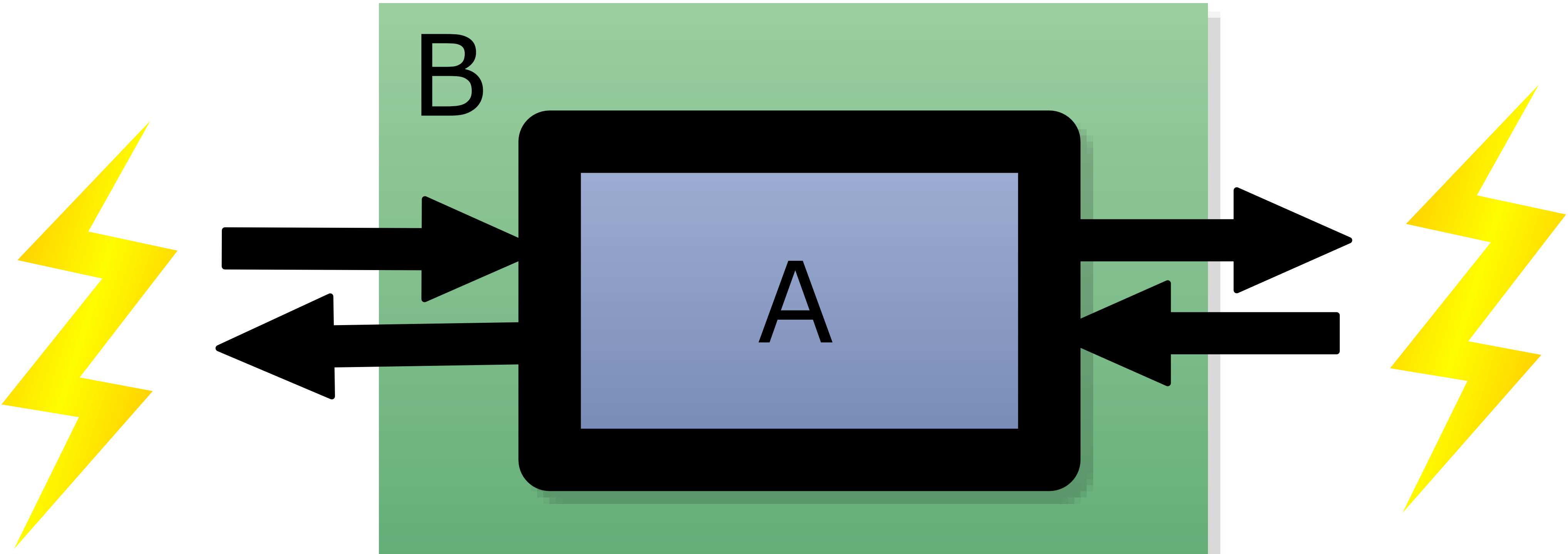
ReentrantLock

```
public class ReentrantLockCounter
implements Counter
{
    private final Lock lock = new ReentrantLock();
    private int c = 0;

    public int increment() {
        lock.lock();
        try {
          return c++;
        } finally {
          lock.unlock();
        }
    }
}
```

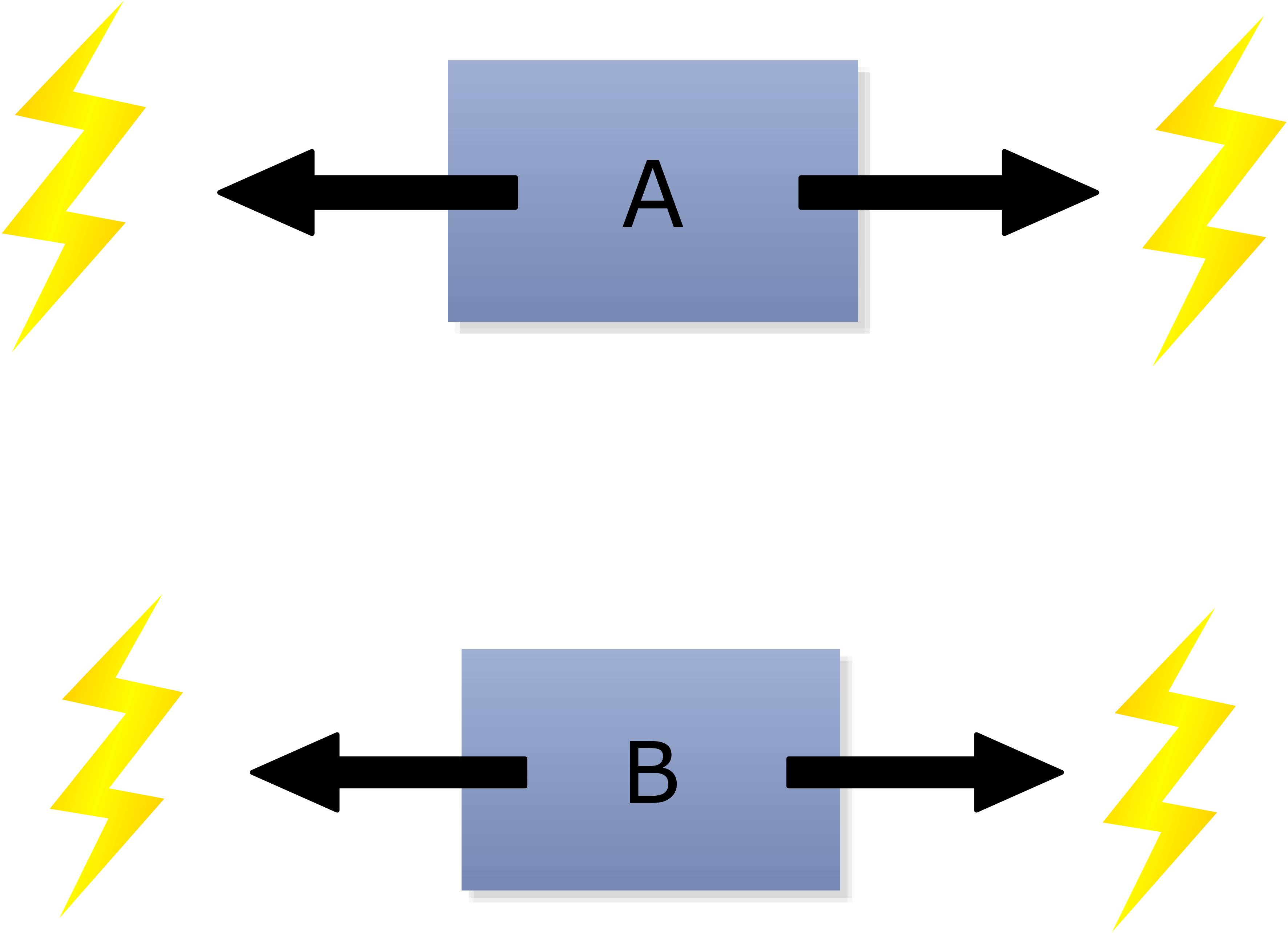
ReentrantReadWriteLock

```
public class ReentrantRWLockCounter implements Counter {
    private final ReadWriteLock lock =
                             new ReentrantReadWriteLock();
    private int c = 0;
    public int increment() {
        lock.writeLock().lock();
        try {
          return c++;
        } finally {
          lock.writeLock().unlock();
        }
    }

    public int read() {
        lock.readLock().lock();
        try {
            return c;
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

Encapsulation



Immutability

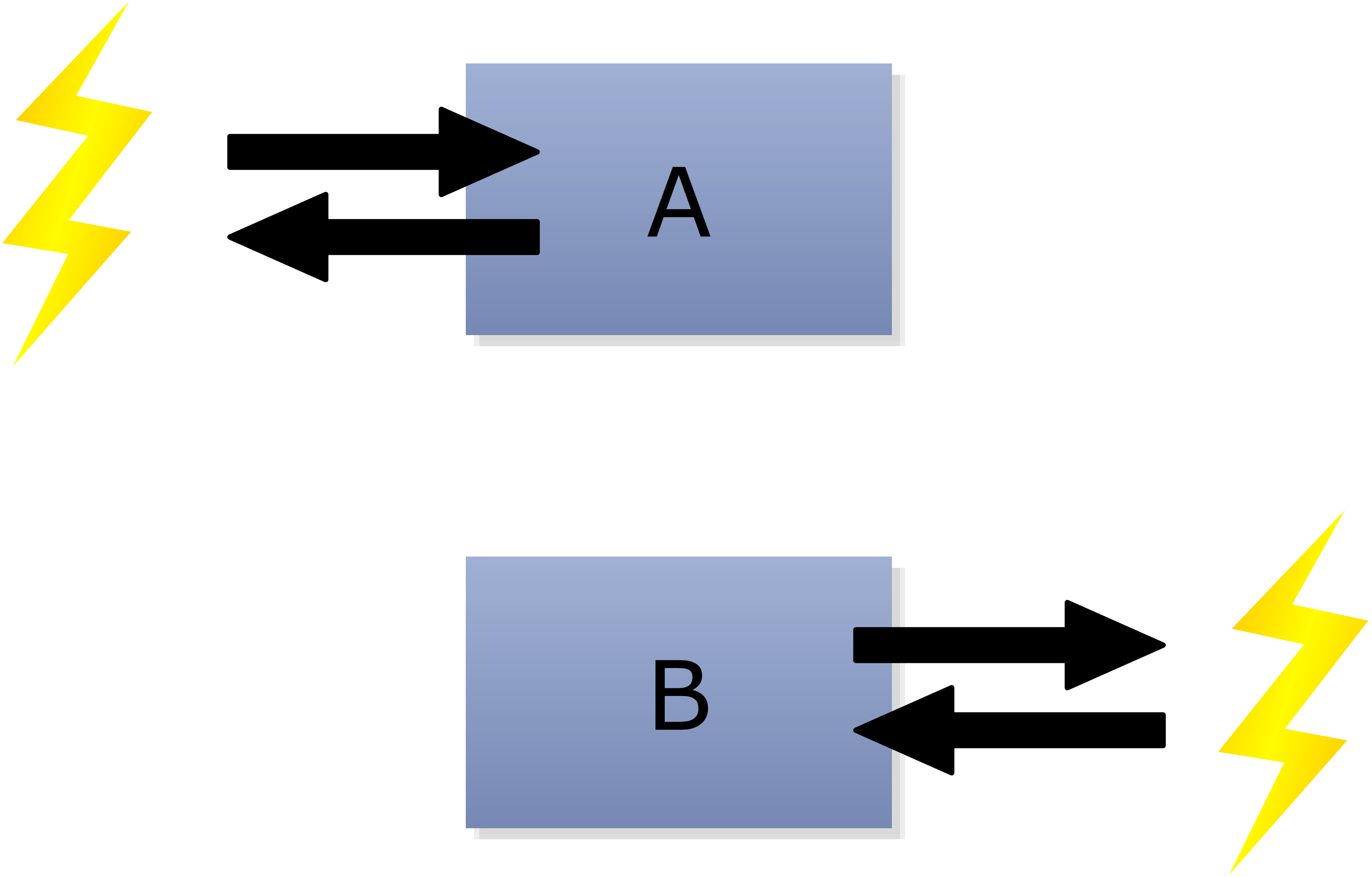


Immutability

Make field final, “mutator” methods return new immutable instances.

```
public class Speed
{
    private final int milesPerHour;

    public Speed(int milesPerHour) {
       this.milesPerHour = milesPerHour;
    }

    public Speed sawCop() {
        return new Speed(this.milesPerHour - 10);
    }
}
```

Thread Confinement



A

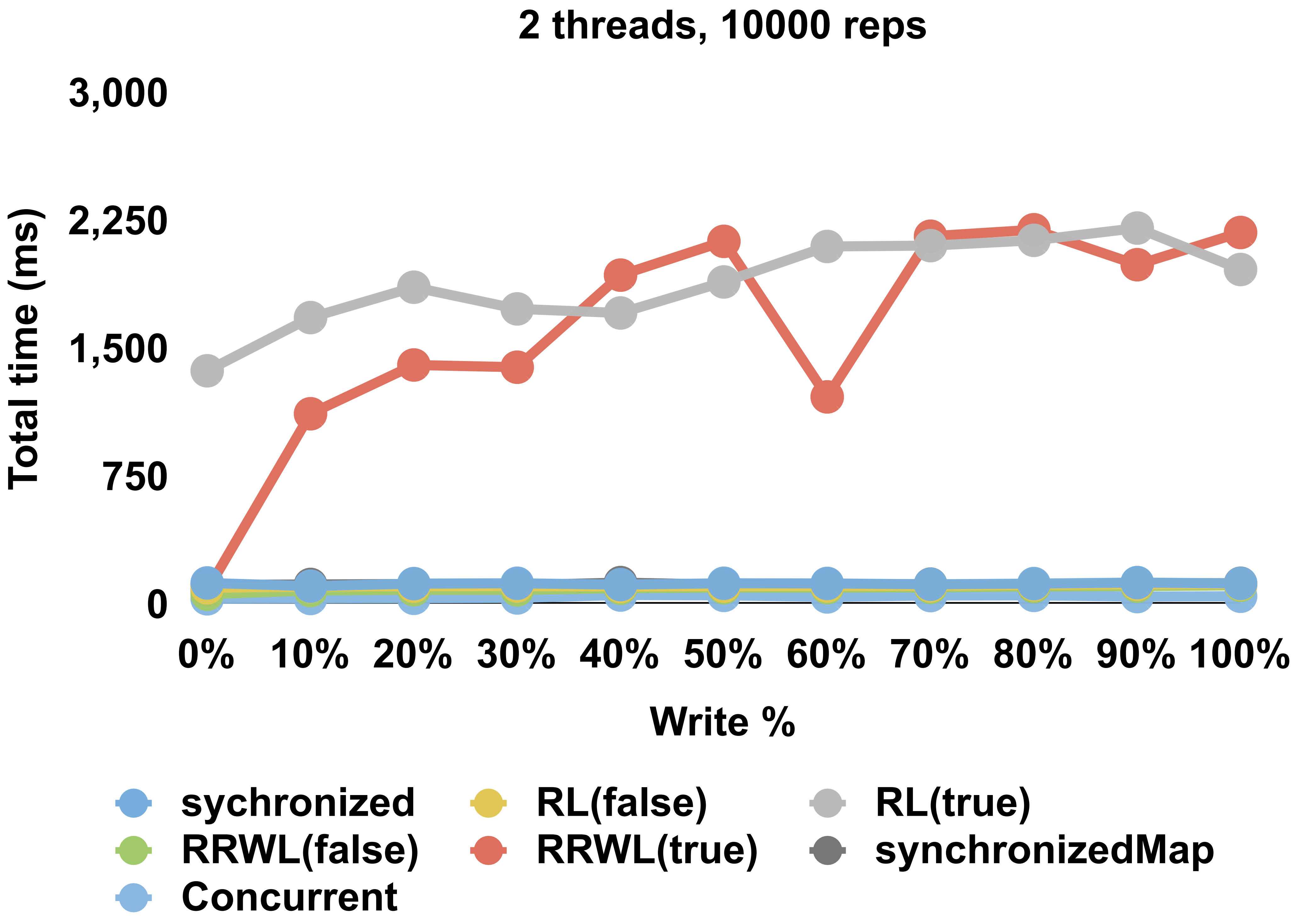
B

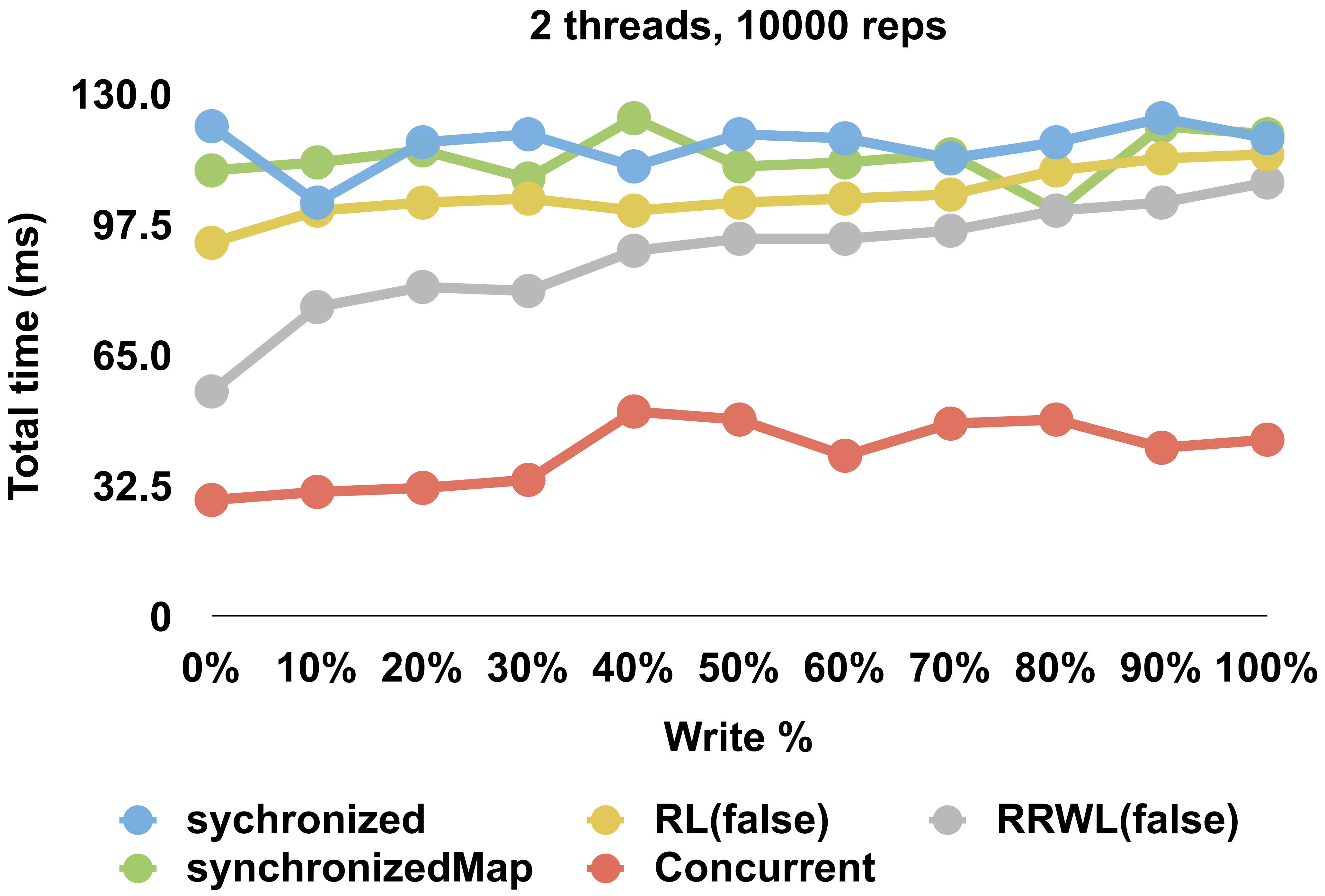
ThreadLocal

ThreadLocal gives every Thread its own instance, so no shared state.

```
public class ThreadLocalCounter implements Counter
{
    private final ThreadLocal<Integer> count =
      new ThreadLocal<Integer>();

    public ThreadLocalCounter() {
        count.set(Integer.valueOf(0));
    }

    public int increment() {
        Integer c = count.get();
        int next = c.intValue() + 1;
        count.set(Integer.valueOf(next));
        return next;
    }
}
```

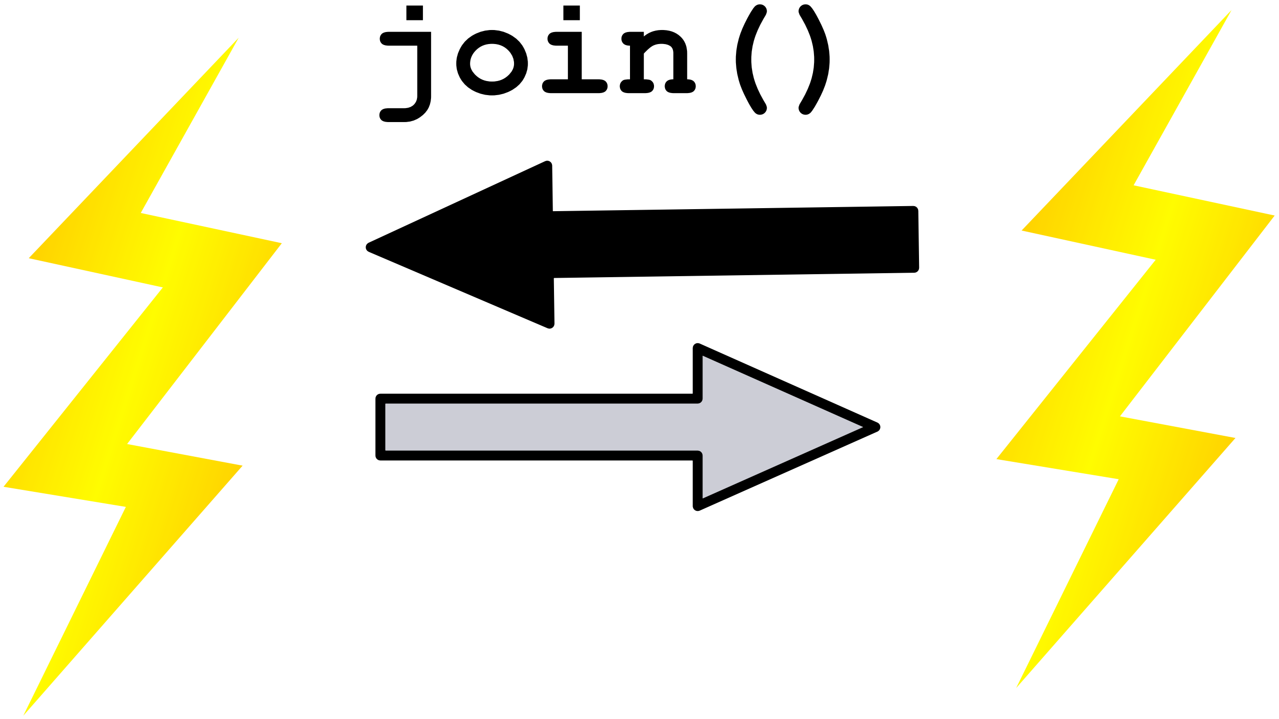
```
code.run()
```


2 threads, 10000 reps
Total time (ms)
3,000
2,250
1,500
750
0
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Write %
sychronized
RL(false)
RL(true)
RRWL(false)
RRWL(true)
synchronizedMap
Concurrent


2 threads, 10000 reps
130.0
97.5
65.0
32.5
0
Total time (ms)
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Write %
sychronized
synchronizedMap
RL(false)
Concurrent
RRWL(false)

Signals

Direct Thread Interaction



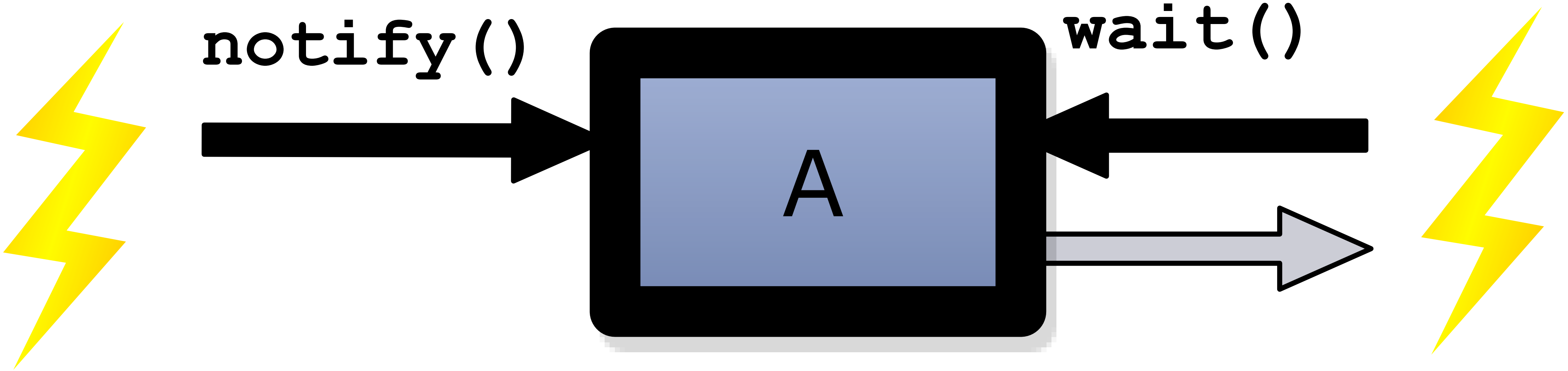
`join()`

join()

join() waits for another Thread to exit - signaling by completion

```
Thread[] threads = new Thread[THREADS];

// start threads doing stuff

// wait for completion
for(int i=0; i<THREADS; i++) {
    threads[i].join();
}
```

Wait / Notify



wait()

- wait() must occur in synchronization
- should occur in loop on the wait condition

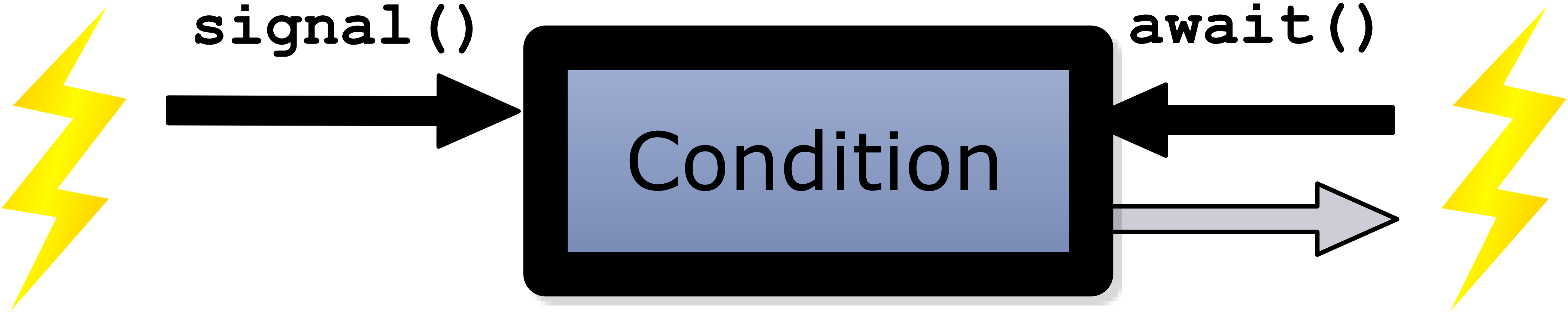
```
synchronized(lock) {
    while(! someCondition) {
        lock.wait();
    }
}
```

notify() / notifyAll()

- notify() / notifyAll() must occur in synchronization

```
synchronized(lock) {
    lock.notifyAll();
}
```

Conditions



Condition waiting

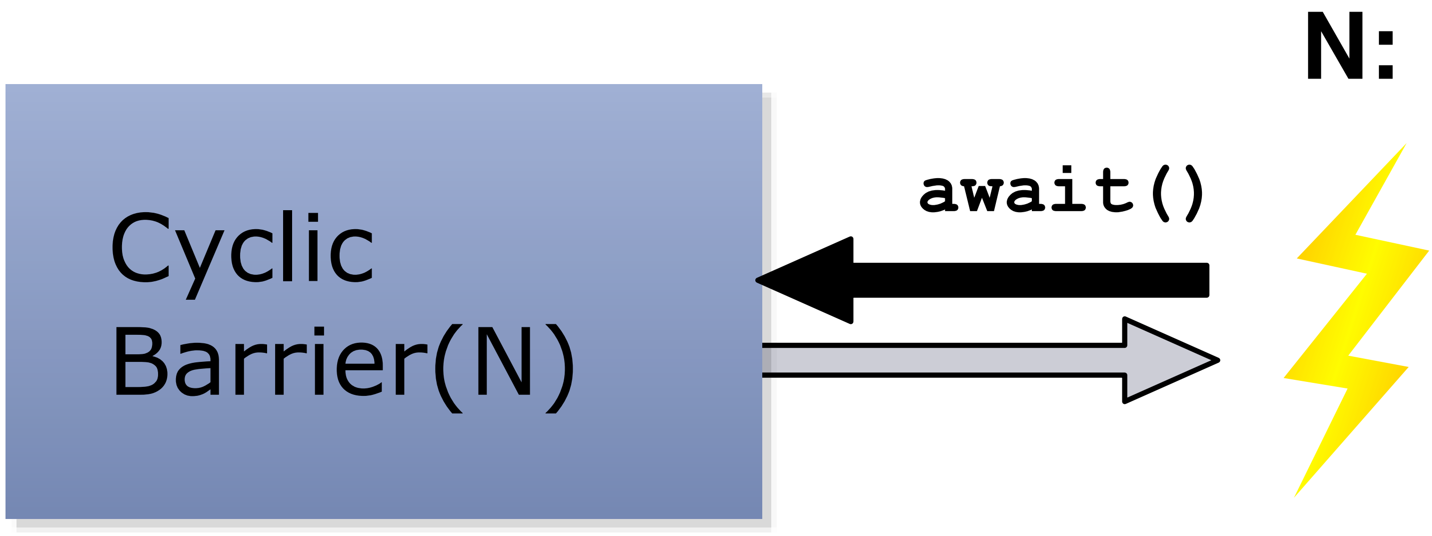
Same as wait/notify but more flexible

```
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();

// wait
lock.lock();
try {
    while(! theCondition) {
        condition.await(1, TimeUnit.SECONDS);
    }
} finally {
    lock.unlock();
}
```

Condition signaling

```
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();

// wait
lock.lock();
try {
    condition.signalAll();
} finally {
    lock.unlock();
}
```

CyclicBarrier



Cyclic Barrier(N)

await()

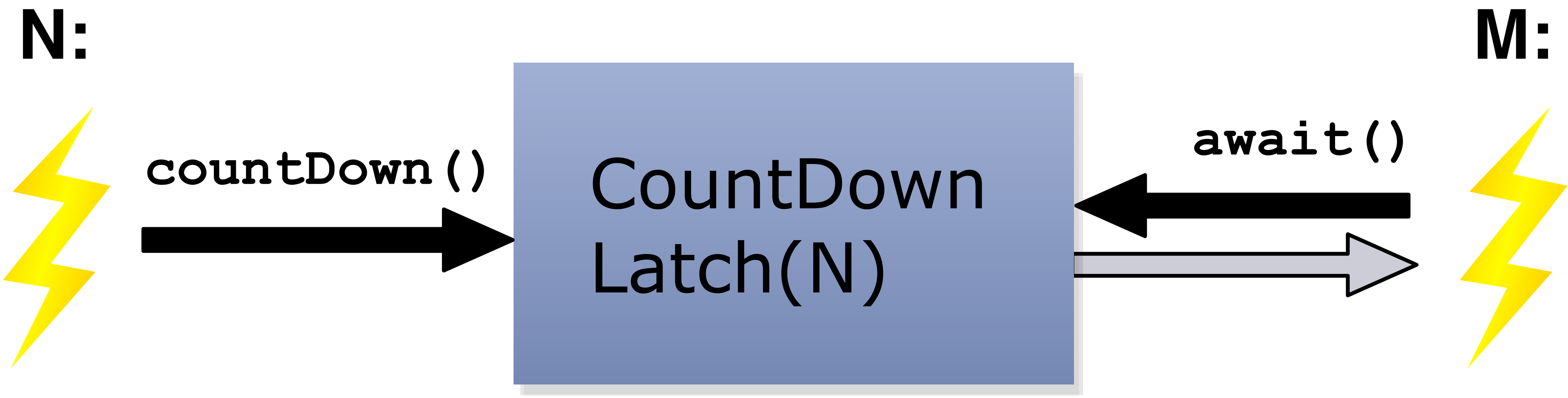
N:

CyclicBarrier

Wait for known # of threads to reach barrier, then release. Can be used multiple times.

```
int THREADS = 5;
CyclicBarrier barrier = new CyclicBarrier(THREADS);

// in thread, wait to start
barrier.await();

// do stuff

// in thread, wait to stop
barrier.await();
```

CountDownLatch



CountDownLatch

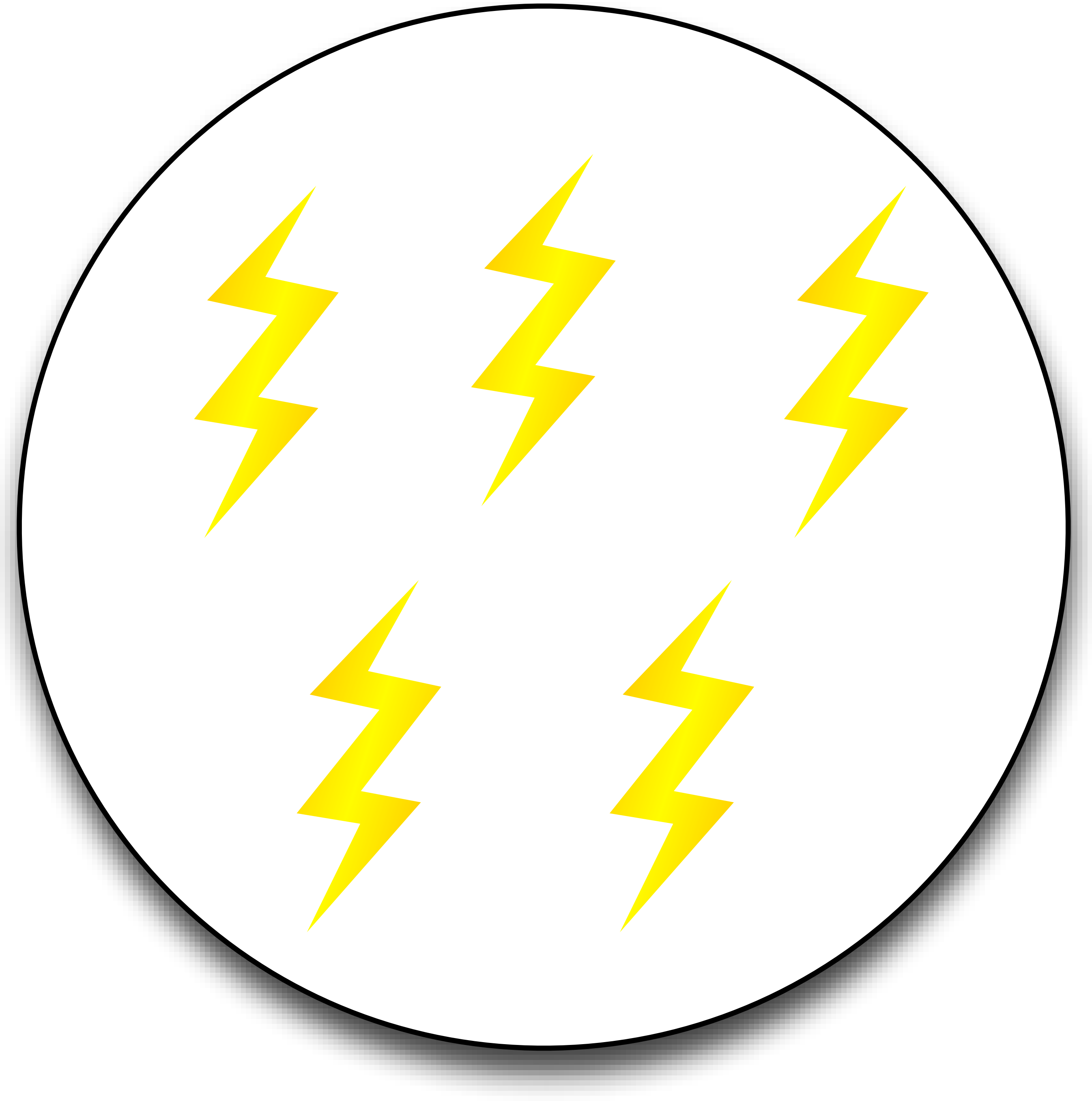
Threads wait for count to reach 0

```
int COUNT = 5;
CountDownLatch latch = new CountDownLatch(COUNT);

// count down
latch.countDown();

// wait
latch.await();
```

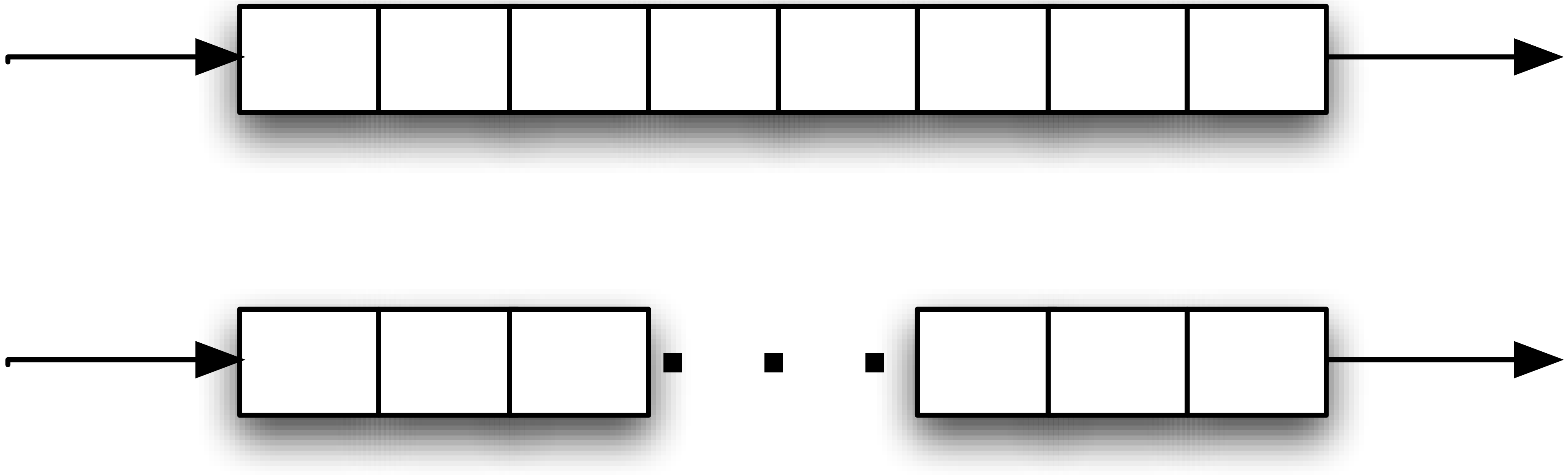
```
code.run()
```

Work

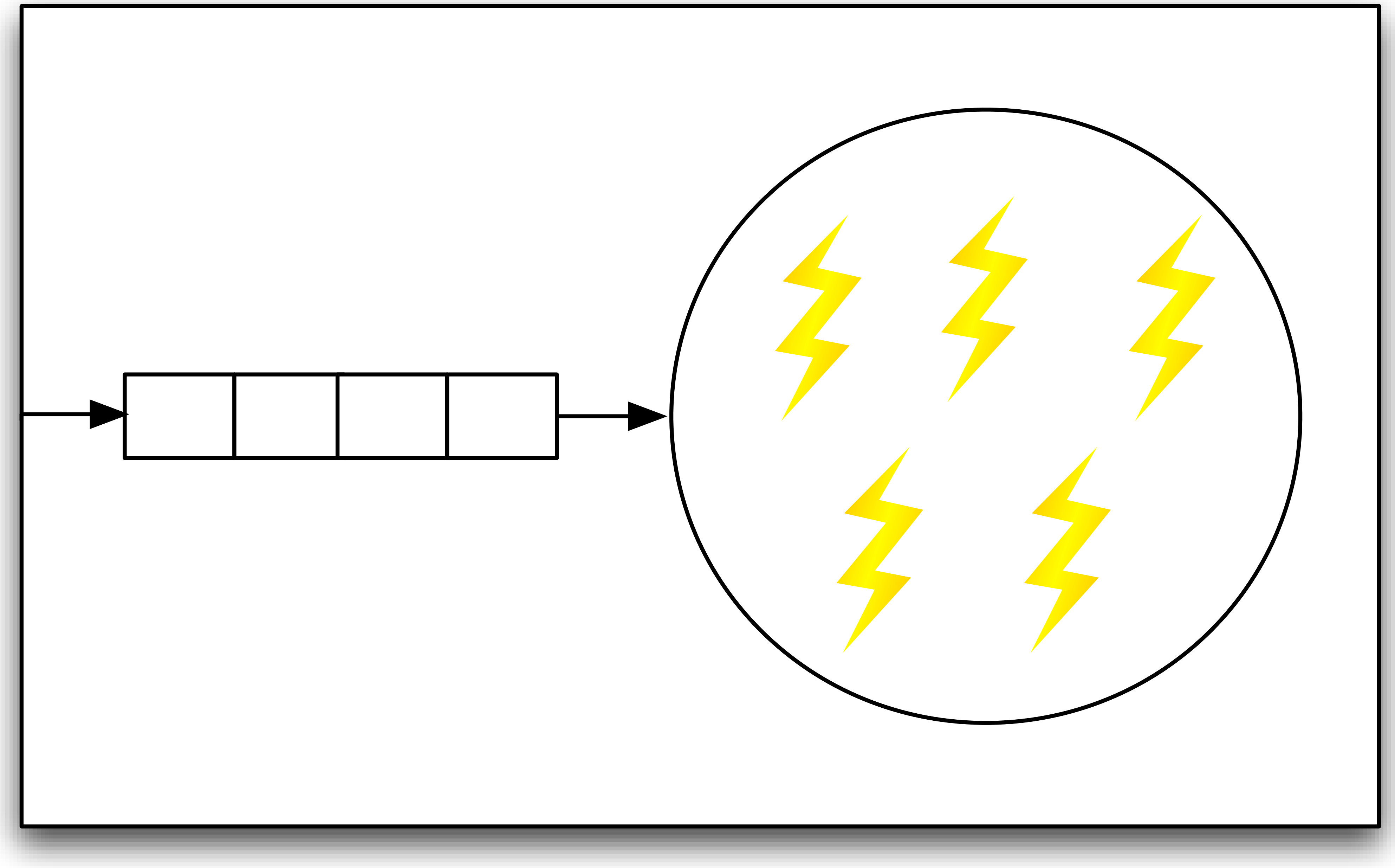
Thread Pools



Queues



ExecutorService

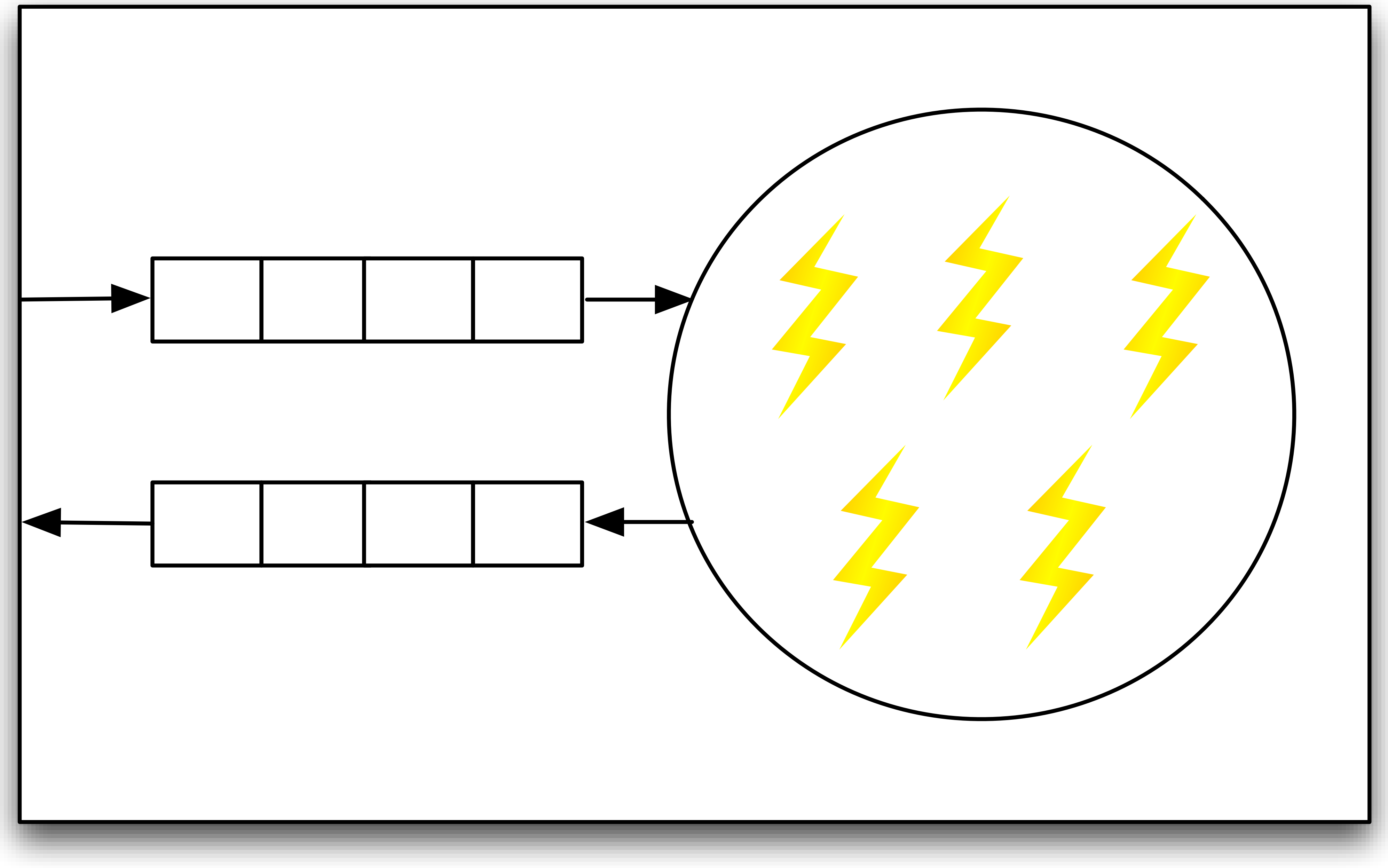


ExecutorService

Executors has helper methods to create different kinds of ExecutorServices backed by thread pools

```
// Create service backed by thread pool
ExecutorService service =
  Executors.newFixedThreadPool(THREADS);

// Define a Work that is Runnable
class Work implements Runnable {...}

// Submit work to the thread pool
service.execute(new Work());
```

CompletionService



CompletionService

CompletionService combines an ExecutorService with a completion queue.

```
// Create completion service backed by thread pool
ExecutorService executor =
    Executors.newFixedThreadPool(THREADS);
CompletionService<Integer> completionService =
    new ExecutorCompletionService<Integer>(executor);

// Submit work
completionService.submit(
    new Callable<Integer>() { .. } );

// Wait for a result to be available
Future<Integer> result = completionService.take();
Integer value = result.get();   // blocks
```

```
code.run()
```

Questions?

Sharing

Work

Signals

Blog: http://tech.puredanger.com

Job: http://terracotta.org

Twitter: http://twitter.com/puredanger