# Thready Thinking

This is just a quick example, touching on some of the mindset for threaded programming. Consider this bit of code which works on a list. Suppose that the list is also used by another thread:

```
public void foo() {
  if (list.size() == 6) {
    System.out.println(list.size());  // you'd think this would always print "6"
  }
}
```

The above code is hard to see correctly because we so accustomed to reading code linearly. Once line N establishes something, it's still true on line N+1. However, memory we share with another thread does not work that way. The existence of another thread modifying the shared memory destroys the coherent, linear view of memory. The above code may not print 6 if some other thread manipulates the list as we are looking at it. Unfortunately, the code will very often print 6 during testing, but there is always the rare possibility that it will print something else.

You can think of a lock as providing a little area of clarity, temporarily blocking out all the other threads changing the memory. While under the lock, things make sense again. Variables are just what they say they are. Here is the same list code, but using the list itself as a lock to protect the list (another lock could be used, so long as all the list-modifying code is consistent about which lock goes with the list).

```
public void bar() {
  synchronized(list) {
    // **** begin coherent view of list
    if (list.size() == 6) {
      System.out.println(list.size());  // prints "6"
    }
  } // **** coherence ends!
}
```

This is the basic approach for threads -- add locking discipline so that all the code that touches shared memory does so under the protection of a lock.

Suppose you have a counting semaphore, called "counter", and there's a need2Permits() method that we only want to run if 2 permits are available, like this:

```
if (counter.getPermits() >= 2) {   // NOTE this code is not quite right
  counter.acquire();
  counter.acquire();
  needs2Permits(); // depends on 2 permits
  ...
```

This code does not work. It has the same problem as the list example above (also, there is no method called getPermits()). The problem with this code is all about the time in between the calls to getPermits() and acquire(). What if some other thread modifies the number of permits just after the call to getPermits()? After the call to getPermits(), everything can change anyway, so the if-statement doesn't really do anything.

The semaphore has an internal locking strategy to protect its permits, and we don't have access to that lock. We rely on the semaphore to deal its locking properly internally, and build our solution just on the simple capabilities the semaphore exposes. One solution is to just go ahead and call acquire() twice with no if-check first. It will block as needed, but when both are done, we definitely have the 2 permits.

```
  counter.acquire();
  counter.acquire();
  needs2Permits();
```

A slight improvement is to use the acquire(int n) variant, which acquires multiple permits in one step:

```
counter.acquire(2);
needs2Permits();
```

Either of the above have the potential problem of using up the permits too soon, while perhaps some other thread needs them. This could be solved with another goAhead semaphore, intended to hold up the 2-credit process until some other part of the program signals by a goAhead.release() that the 2-credit process should starting trying to acquire the 2 credits:

```
goAhead.acquire();   // First block on this
counter.acquire(2);  // Now go ahead and grab the credits
needs2Permits();
```

Trying to get the permits value out of the semaphore will never work. Instead, we build a solution up out of the fundamental acquire() service that the semaphore provides.