SOLUTION NOTES

Further Java 2001 Paper 4 Question 3 (TLH)

(*a*) Describe how mutual-exclusion locks provided by the `synchronized` keyword can be used to control access to shared data structures. In particular you should be clear about the behaviour of concurrent invocations of different synchronized methods on the same object, or of the same synchronized method on different objects.

- Two kinds of syntax: either as a modifier on a method definition or as a 'synchronized block', e.g. `synchronized (ref) { .. }` where `ref` is an object reference.
- A lock is associated with each object. Hence only one synchronized method may be called concurrently on a given object. The same synchronized method may be called concurrently on different objects.

(*b*) Show how to start 2 threads, each executing the `run` method of this class.

```
public static void main(String args[]) {
   new Thread (new Example)).start ();
   new Thread (new Example)).start ();
}
```

(*c*) When this program is executed only one of the `count` fields is found to be incrementing, even though threads are scheduled pre-emptively. Why might this be?

The second thread happens to never be scheduled during the brief intervals at which the first thread has released the mutex.

(*d*) Define a new class `FairLock`. Each instance should support two methods, `lock` and `unlock`, which acquire and release a mutual exclusion lock such that calls to `unlock` never block the caller, but will allow the longest-waiting blocked thread to acquire the lock. The lock should be recursive, meaning that the thread holding the lock may make multiple calls to `lock` without blocking. The lock is only released when a matched number of `unlock` operations have been made.

(*e*) A queue class to maintain a list of waiting threads:

```
  class Queue {
    Thread waiter;
    Queue  next;
    public Queue (Thread waiter, Queue next) {
      this.waiter = waiter;
      this.next   = next;
    }
  }
```

The lock class itself:

```
  class FairLock {
    Thread owner = null;
    Queue  oldest_waiter;
    Queue  youngest_waiter;
    int    count;

    public synchronized void lock() throws InterruptedException {
      Thread caller = Thread.currentThread ();
      if (owner == null) {
        count = 1;
        owner = caller;
      } else if (caller == owner) {
        count ++;
      } else {
        Queue me = new Queue (caller, null);
        if (youngest_waiter != null) {
          youngest_waiter.next = me;
        } else {
          oldest_waiter = me;
        }
        youngest_waiter = me;
        do { wait (); } while (oldest_waiter != me);
        oldest_waiter = oldest_waiter.next;
        if (youngest_waiter == me) {
          youngest_waiter = null;
        }
      }
    }

    public synchronized void unlock () {
      count --;
      if (count == 0) {
        notifyAll ();
      }
    }
  }
```