```
// This is a demonstration of how race conditions can get you
// in concurrent code. It turns out to be much harder to demonstrate
// than it seems it should be, which also provides a cautionary tale:
// just because code seems to work, if you didn't follow the rules
// it is nevertheless broken.
//
// In testing on my home machine with NetBeans 3.6 (and whatever
// compiler/runtime it came with)
// it's hard to make it fail.
class Race {
    static long x = 0;
    static int limit = 100000;
    static Barrier b1 = new Barrier(2);
    static Barrier b2 = new Barrier(2);
    // Barrier is a class implementing so-called barrier synchronization.
    // The idea is to provide a point where all participating threads must
    // arrive before any of them proceed to the next instruction.
    //
    // new Barrier(numThreads) creates a barrier object that expects
    // numThreads
    // arrivals before any pass. If b is a barrier, a thread calls b.synch()
    // to
    // wait for the other threads.

    static class Barrier {
        int arrived = 0;
        int passed = 0;
        int numThreads;

        Barrier (int numThreads) {this.numThreads = numThreads;}
        synchronized boolean synch(int id) {
            arrived += 1;
            while (arrived < numThreads) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    return false;
                }
            }
            // at this point arrived == numThreads in all participants.
            // But the first to get here has to awaken the others.
            if (passed==0) {
                notifyAll();
            }
            // Now must arrange to reset the counters. This can only be done
            // when all have passed the barrier point.
            passed += 1;
            // When all have passed, reset the counters.
            if (passed==numThreads) {
                arrived = 0;
                passed = 0;
            }
            return true;
        }
    }

     // Racer class represents the individual threads that are going to participate
     // in the race.
         static class Racer extends Thread {
             int id;
             Racer (int id) {
                 this.id = id;
         }
         // run() is the guts. Note that with the body of the loop being just x
         // = x+1 NO failures are observed (though the code
```

```
        // is still wrong. Even with this complicated loop body, many
        // thousands of iterations of the outer loop are needed before
        // an error occurs.
        //
        // It is also of interest that the observed failures are that x mod
        // 100000 == 0 but x mod 20000 == 10000. That is, it appears that
        // the code for the loop is loading/storing x from shared memory only
        // once per iteration of the outer loop. It would not have to be this
        // way; other misbehaviors are possible.

        public void run() {
            int i;
            int j;
            for (i=0; i<limit; i++) {
                for (j=0; j<10000; j++) {
                    long y = x;
                    x = 3 * x;
                    x = x - y;
                    x = x - y;
                    x = x + 1;
                }
                if (!b1.synch(id)) {
                    System.out.println("Synch 1 interrupted");
                    return;
                }
                if (x%20000 != 0) {
                    System.out.println(id + ": Failed with x=" + x);
                }
                if (!b2.synch(id)) {
                    System.out.println("Synch 2 interrupted");
                    return;
                }
            }
            System.out.println("Racer number " + id + " finished. x=" + x);
        }
     }
    public static void main(String argv[]) {
        Racer t1 = new Racer(0);
        Racer t2 = new Racer(1);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {}
    }
}
```