1. Multithreading, Data Races, and Deadlock
   a. For the following program, give two schedules under which the final value of $i$ differs in the two schedules. Give the schedule as a list of line numbers, and in each case, also give the final value of $i$.

   ```java
   l = new ReentrantLock();
   m = new ReentrantLock();
   i = 0

   Thread 1
   1. l.lock();
   2. i = 3;
   3. l.unlock();

   Thread 2
   4. m.lock();
   5. i = i + 1;
   6. m.unlock();

   Since `l` and `m` are different locks, they do not provide any mutual exclusion in this example, and the threads may be interleaved arbitrarily.

   Some example solutions:
   1, 2, 3, 4, 5, 6 - i = 4
   1, 4, 5, 2, 3, 6 - i = 3
b. For the following program, give one schedule under which there will be a
deadlock, and give one schedule under which there will not be a deadlock. Give
the schedule as a list of line numbers.

```java
l = new ReentrantLock();
m = new ReentrantLock();
n = new ReentrantLock();

Thread 1
1. l.lock();
2. m.lock();
3. m.unlock();
4. l.unlock();

Thread 2
5. m.lock();
6. n.lock();
7. n.unlock();
8. m.unlock();

Thread 3
9. n.lock();
10. l.lock();
11. l.unlock();
12. n.unlock();

Some example solutions:

No deadlock - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Deadlock - 1, 5, 9
Because thread 1 holds l, thread 2 holds m, and thread 3 holds n, so none of the threads can make progress
c. Using language notation similar to above, write a program that has no data races, but nonetheless may produce different output under different schedules.

There are many possible answers, such as:

```java
l = new ReentrantLock();
i = 0;

Thread 1
l.lock();
i = 2;
l.unlock();

Thread 2
l.lock();
i = 3;
l.unlock();

Here the final value of i is either 2 or 3, but there is no data race.
```

d. List all possible outputs from the following program. Indicate next to each possible output whether all threads complete at the end, or, if they do not, which threads remain blocked.

```java
l = new ReentrantLock();
c = l.newCondition()

Thread 1
l.lock();
System.out.print("a ");
c.await();
System.out.print("b ");
l.unlock();

Thread 2
l.lock();
System.out.print("c ");
c.signalAll();
System.out.print("d ");
l.unlock();

Possible outputs:

    a c d b - all threads complete
    c d a - thread 1 remains blocked, waiting to be woken up
```
2. Multithreading Code
   a. Using Java Conditions, you must implement a synchronization construct called MyBarrier. A MyBarrier object is created with a certain value n. When a thread calls the method enter(), it enters the barrier and blocks until a total of n threads have entered the barrier. When the n<sup>th</sup> thread enters the barrier, all the threads waiting at the barrier wake up and unblock, and the n<sup>th</sup> thread continues without blocking. When a thread calls the method reset(), the barrier is reset so that it starts fresh in counting up to n (i.e., n more threads must enter the MyBarrier). You may start by modifying the following code fragment:

```java
public class MyBarrier {
    public void MyBarrier (int n) { … }
    public enter( ) { … }
    public reset( ) { … }
}

public class MyBarrier {
    int num; // shared read-only data
    int current = 0; // shared modifiable data
    Lock lock = new ReentrantLock();
    Condition ready = lock.newCondition();

    public MyBarrier (int n) {
        num = n;
    }

    public void enter( ) throws InterruptedException {
        lock.lock(); // prevent data race on current
        current++; // incr # of threads at barrier
        if (current == num) { // enough threads at barrier
            ready.signalAll(); // wake up other threads
        } else {
            while (current < num) { // wait for more threads to enter
                ready.await(); // sleep until enough threads enter
            } // use while ( ) in case reset( ) called
        }
        lock.unlock();
    }

    public void reset( ) {
        lock.lock(); // prevent data race on current
        current = 0;
        lock.unlock();
    }
}
```
b. Implement MyBarrier using Ruby monitors.

```ruby
require "monitor.rb"

class MyBarrier
  def initialize n
    @num = n
    @current = 0
    @myLock = Monitor.new
    @myCondition = @myLock.new_cond
  end

  def enter
    @myLock.synchronize {
      @current = @current + 1
      if @current == @num then
        @myCondition.broadcast
      else
        @myCondition.wait_while { @current < @num }
      end
    }
  end

  def reset
    @myLock.synchronize {
      @current = 0
    }
  end
end
```

c. Write a Ruby program that creates a barrier for 2 threads, then creates 2 threads that each print out “hello”, enters the barrier, then prints out “goodbye”.

```ruby
bar = MyBarrier.new 2

t1 = Thread.new {
  puts "hello"
  bar.enter
  puts "goodbye"
}

t2 = Thread.new {
  puts "hello"
  bar.enter
  puts "goodbye"
}
```