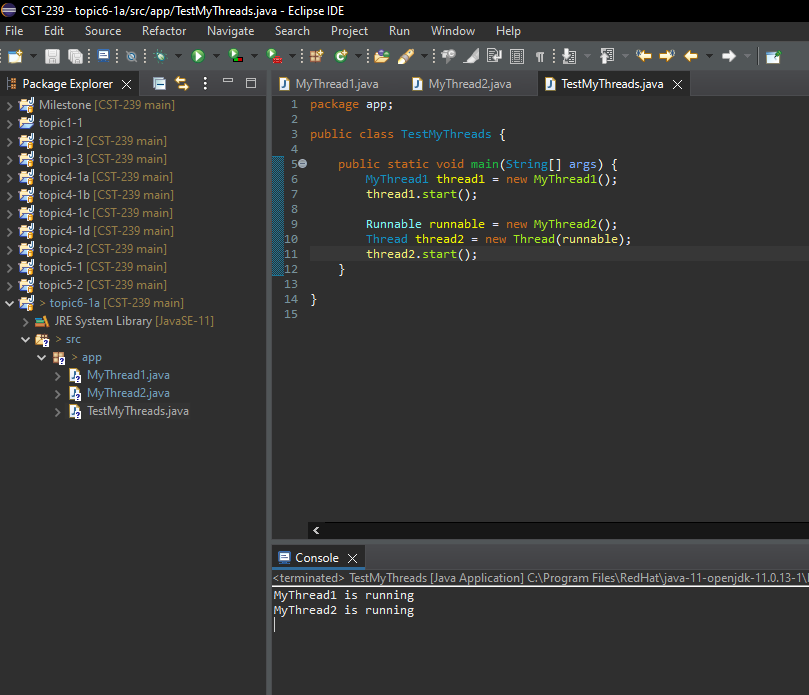
CST-239 Activity 6 Guide

Zachary Pahl

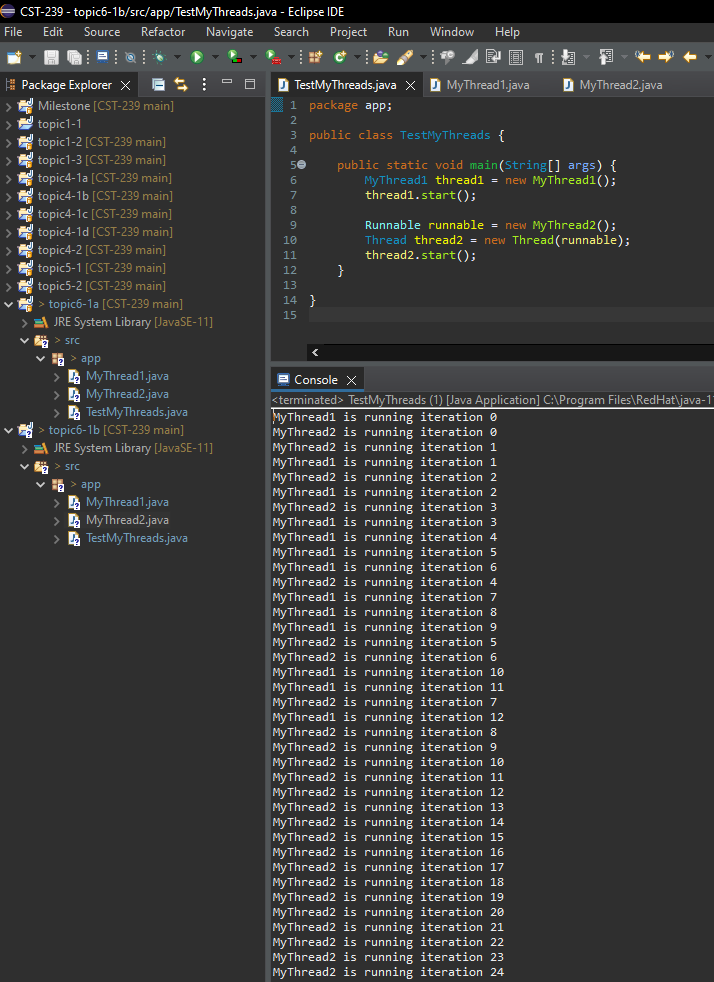
4/17/2022

Part 1a

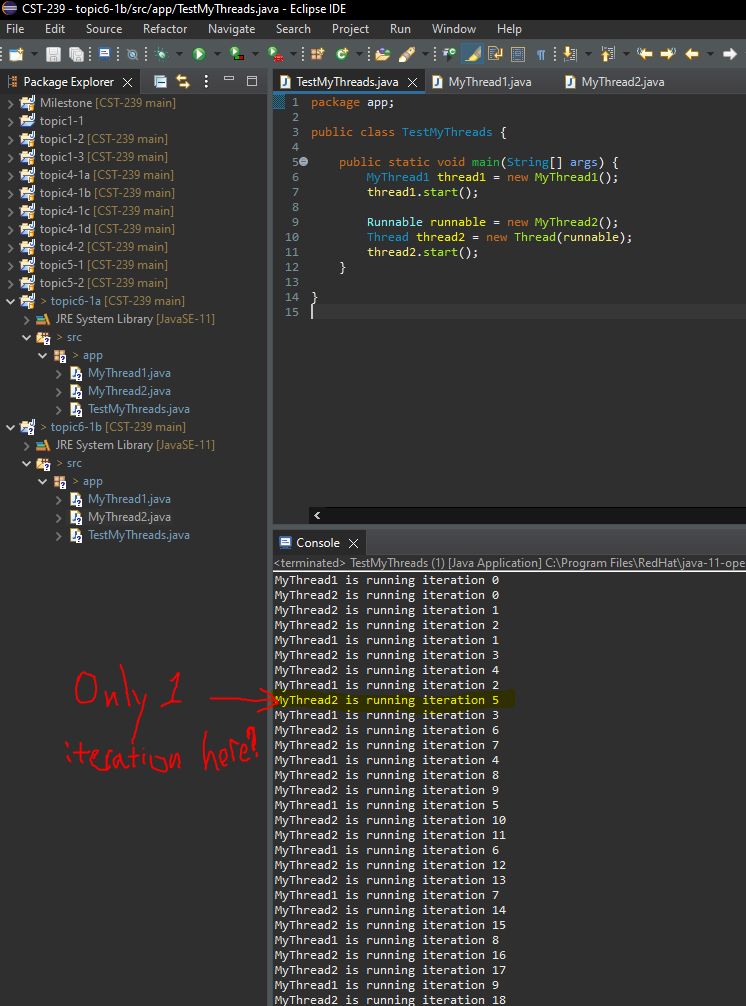


These threads run by creating objects of class MyThread1 and MyThread2. The thread1 object’s class extends Thread, and therefore has the start() method. In our code, we made our own run() method that simply prints a message to the console. The thread2 Thread object was created by first making a Runnable object with MyThread2 class (which has the overridden run() method that also simply displays a message in the console) and passing that to the Thread constructor.

Part 1b

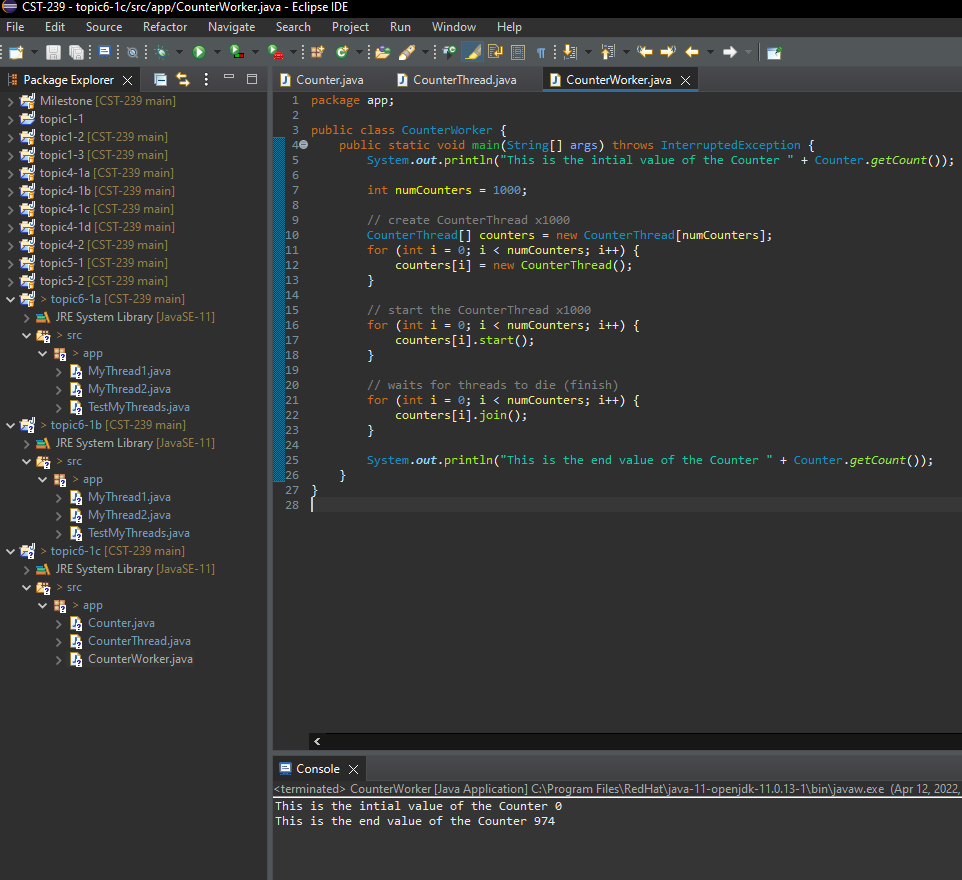


My output shows the threads were executing concurrently, just like how the textbook describes it. The two threads are created for each task respectively, then the CPU has to share the console while it “takes turns” to print out the desired line per our run() method defined in each task. Running the TestMyThreads main() method multiple times resulted in slightly different outputs on the order of MyThread1 vs MyThread2 (MyThread1 always printed first though, although I didn’t take a thorough/large sample size number of runs).

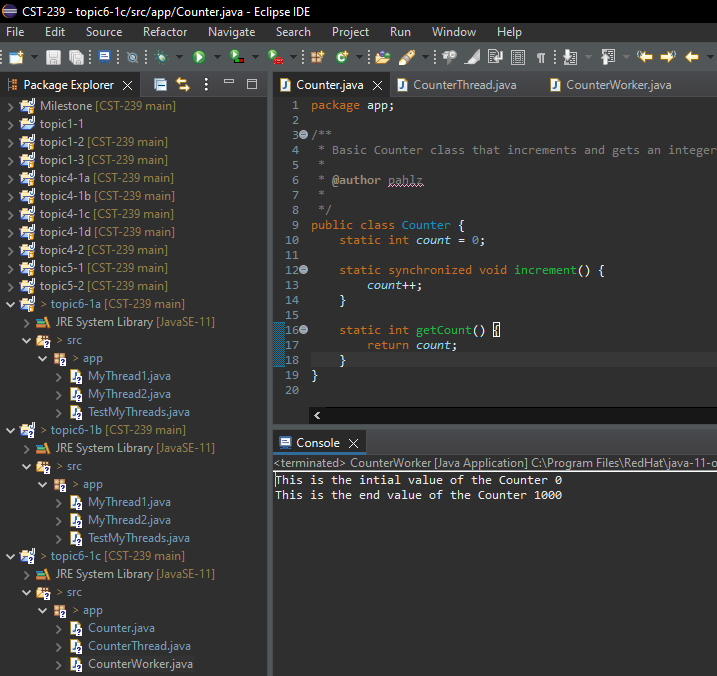


The Thread.sleep(1000) and Thread.sleep(500) in each run() method dictate to the CPU how fast to execute each thread. Thus, MyThread2 will obviously be looping twice as fast (or MyThread1 will be sleeping twice as long on each loop). So even though each thread is technically running concurrently, due to the sleep command, they’re printing “in turn”.

Part 1c

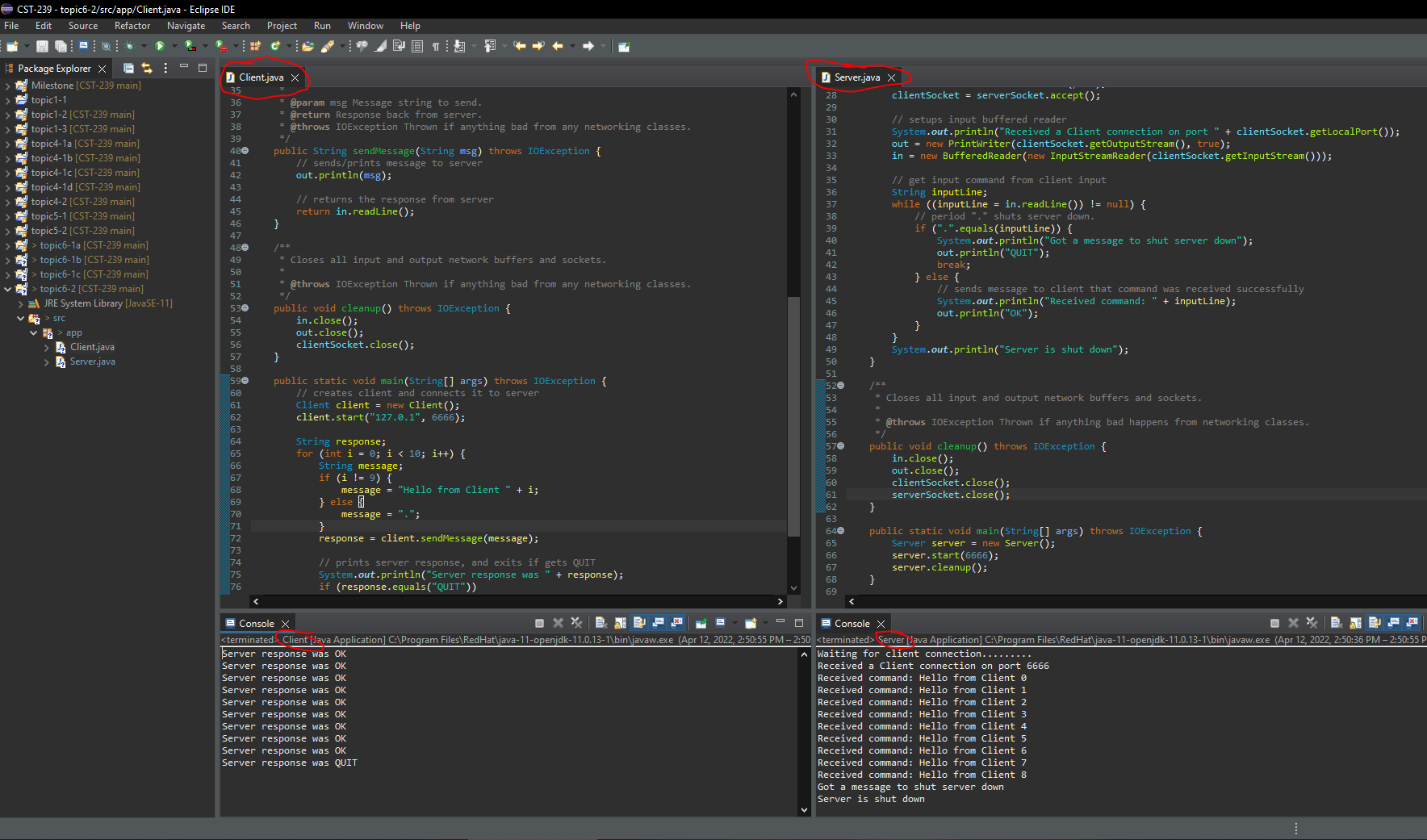


It’s expected that the final value of the Counter should be 1000, as there were 1000 threads created, each incrementing Counter by 1. However, after multiple runs, the end value ends around 980. This is due multiple threads accessing the same variable concurrently and causing a conflict. The shared Counter data has become corrupted due to there being a race condition (Counter is not “thread-safe” class).



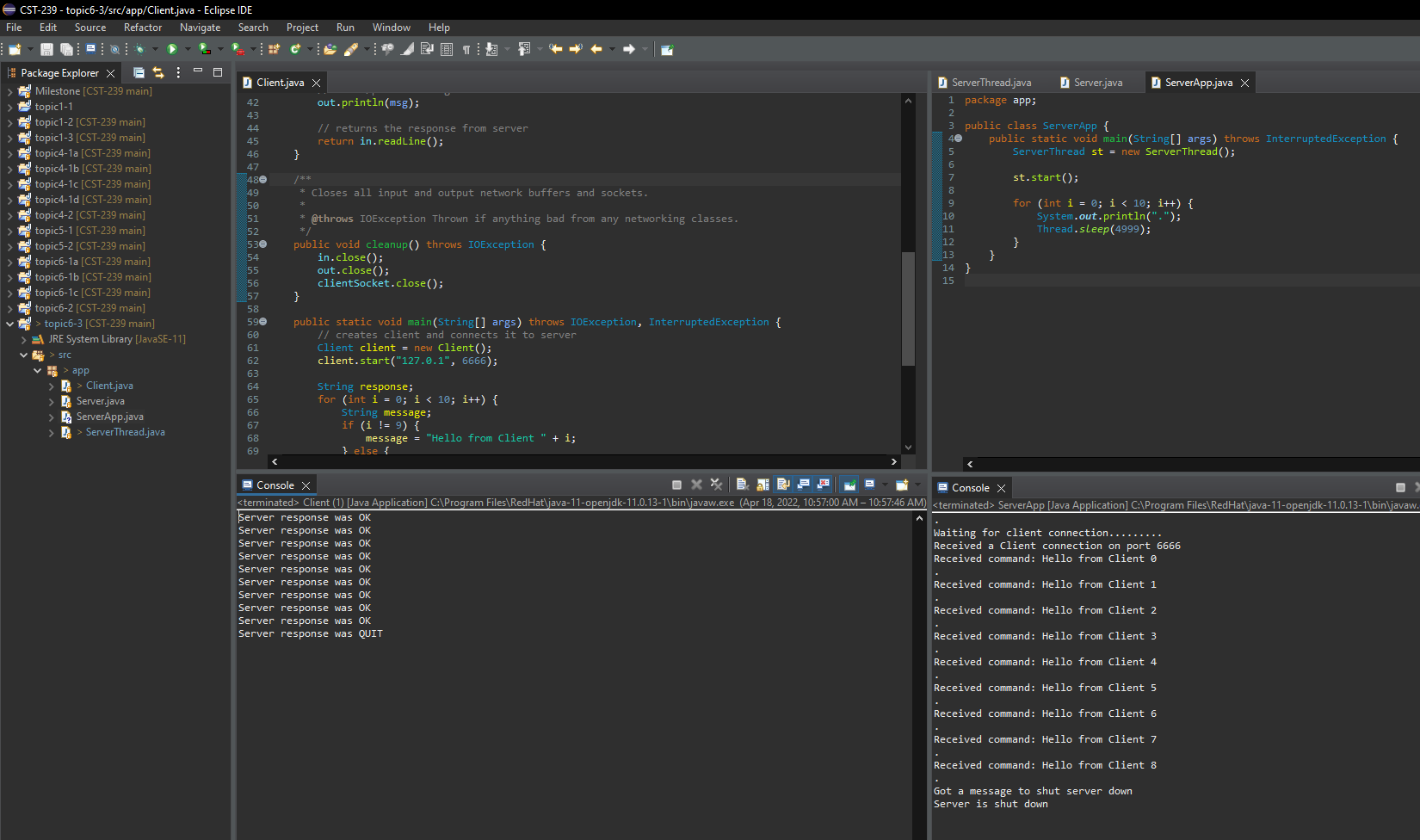
This fixes the above problem by simply adding the synchronized keyword to the increment() method in Counter class. This fixes the problem because the synchronized keyword causes the method to acquire a lock on the Counter, and the multiple threads will now appropriately “take turns” to access and modify the Counter. This particular solution changes the increment() method to make the entire method synchronized.

Part 2



The Server class and Client class create a network client/server application. This is first done by starting the Server on a port number. The ServerSocket accepts a client connection and stores it in a socket for the client. Then the server sets up a buffered reader and print writer to display sent and received messages between itself and the client. The Client class may now start() and connect to the port number of the server (and create a buffered reader and writer). The network connection is now made, and our Client class has a sendMessage method that demonstrates the connection via printing to the console. NOTE: Both classes have separate consoles they are printing to! Additionally, both classes have a method cleanup() that closes all network ports, readers, writers, etc.

Part 3



Java Threads improved the design of this server because it allowed the Server to run asynchronously to the client running the server. This is done because the ServerApp creates its own ServerThread to run its task on.