

**Java Program Analysis**

Matthew S McCormick

CSU Global

Section C

CSC450

Farhad Bari

Even though the threads in my program don't modify any shared objects or variables, concurrency still results in some performance challenges. The biggest one I found is that when I first let my CountUp and CountDown threads run, they ended up printing to the screen at the same time unpredictably. This made following which thread was counting up and which was counting down confusing. To solve this problem I created a mutex object and passed it to both of the threads on their creation. I then made synchronized blocks that were synchronized to the mutex in each of the threads. This made it so the CountDown thread would wait until CountUp was done printing to the screen before counting down.

My threads don't share any scope other than the mutex object, which is not modified by the threads. There is also no way for the counting while loops to get stuck without exiting the synchronized block. This means there isn't a risk of deadlock where threads wait on each other forever. There is only one way entering and exiting each of the thread's run function, so the synchronized blocks are always executed and exited in the same way.

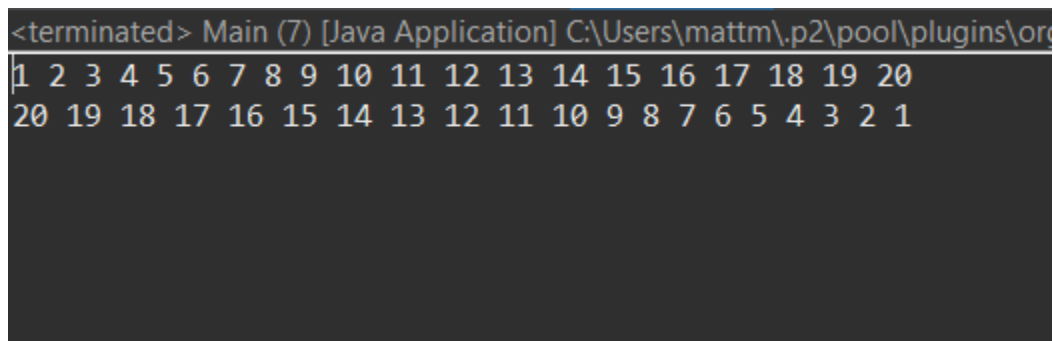
I don't store any strings in variables in my program. However, I do print strings to the screen using the `System.out.print()` method. The method is generally regarded as secure, but it is known to be somewhat inefficient as it is treated as an I/O operation. However, it is efficient enough for the purposes of this program. When multithreading, I found that it is recommended to put `System.out.print()` in synchronized blocks for consistent outputs. Java strings aren't null terminated, and instead rely on storing their length to know where the end of a string is. The “ “ string I use when printing the count has a length of 1 for example.

The data types used in my program are shorts, strings, functions, threads, and objects. I decided to use a short to keep track of the count in my threads, as it can store numbers between -32,768 and 32,767. The count is only ever between 0 and 20 so this is more than sufficient, and will never lead to any integer overflows. The main classes I use are the count classes which extend threads, and the object class used for the mutex. These classes are both automatically freed from memory after the program has executed, as threads are destroyed and

readied for garbage collection upon their run methods being completed. The CountUp and CountDown run functions must be made public in order to extend the Thread class. I made the count variable and mutex objects private, as they should only be accessible by the classes they were declared in, or those they were passed to directly.

### Figure 1

Output of Counting Program



```
<terminated> Main (7) [Java Application] C:\Users\mattm\p2\pool\plugins\or
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

### Main class

```
final class Main {

    // This mutex will be passed to both threads to synchronize them
    private static Object mutex = new Object();

    public static void main(String[] args) {

        // Creating threads and passing start value + mutex
        CountUp t1 = new CountUp((short)0, mutex);
        CountDown t2 = new CountDown((short)20, mutex);

        // Starting threads
        t1.start();
```

```

        t2.start();
    }
}

```

### CountUp Class

```

final class CountUp extends Thread {

    // Variable is only used in this class so it is made private
    private short count;

    // Holds mutex to synchronize with
    private Object mutex;

    // Constructor for thread
    CountUp(short count, Object mutex) {
        this.count = count;
        this.mutex = mutex;
    }

    public void run() {
        // Entering critical region
        synchronized(mutex) {
            // Loops until count is 20
            while (count < 20) {
                // Incrementing count
            }
        }
    }
}

```

```

        count++;

        // Printing out count
        System.out.print(count + " ");

    }

    // Printing out new line
    System.out.println();

}

}

}

```

### **CountDown Class**

```

final class CountDown extends Thread {

    // Variable is only used in this class so it is made private
    private short count;

    // Holds mutex to synchronize with
    private Object mutex;

    // Constructor for thread
    CountDown(short count, Object mutex) {
        this.count = count;
        this.mutex = mutex;
    }
}

```

```
public void run() {  
    // Entering critical region  
    synchronized (mutex) {  
        // Loops until count is 0  
        while (count > 0) {  
            // Printing count  
            System.out.print(count + " ");  
            // Decrementing count  
            count--;  
        }  
    }  
}
```