Final Portfolio

Colorado State University Global

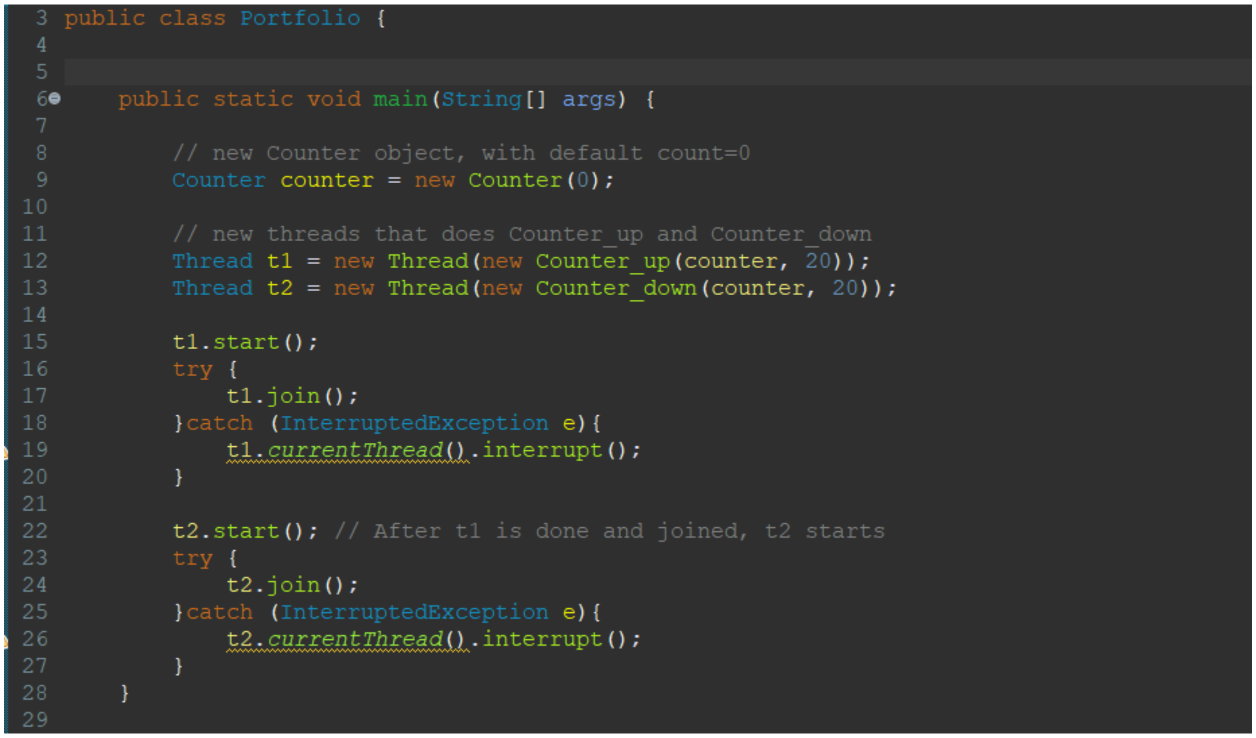
CSC450 Programming 3

Module 8

Student Name : Minhyun Sung

Part 1. Java Application

I made an application that uses two threads in Java, with one thread counting up to 20, and another counting down to 0. Below is the screenshot of my main method in Java.



```java
public class Portfolio {


    public static void main(String[] args) {

        // new Counter object, with default count=0
        Counter counter = new Counter(0);

        // new threads that does Counter_up and Counter_down
        Thread t1 = new Thread(new Counter_up(counter, 20));
        Thread t2 = new Thread(new Counter_down(counter, 20));

        t1.start();
        try {
            t1.join();
        }catch (InterruptedException e){
            t1.currentThread().interrupt();
        }

        t2.start(); // After t1 is done and joined, t2 starts
        try {
            t2.join();
        }catch (InterruptedException e){
            t2.currentThread().interrupt();
        }
    }

```

There are three ways to make threads in Java. Firstly, by using a class that inherits 'Runnable', or extends 'Thread'. Below is the example of method 1.

```
// Method 1. By using a class that inherits 'Runnable'.

Thread t1 = new Thread(new class_name); // class

Static class class_name implements Runnable{

        @Override

        Public void run(){
```

```
                    // Define the method here

          }

}
```

Secondly, we can start a thread object that implements the ‘Runnable’ interface.

```
// Method 2.
Thread t2 = new Thread(new Runnable(){
          @Override

          Public void run(){

                    // things you want to do with the thread

          }

});
```

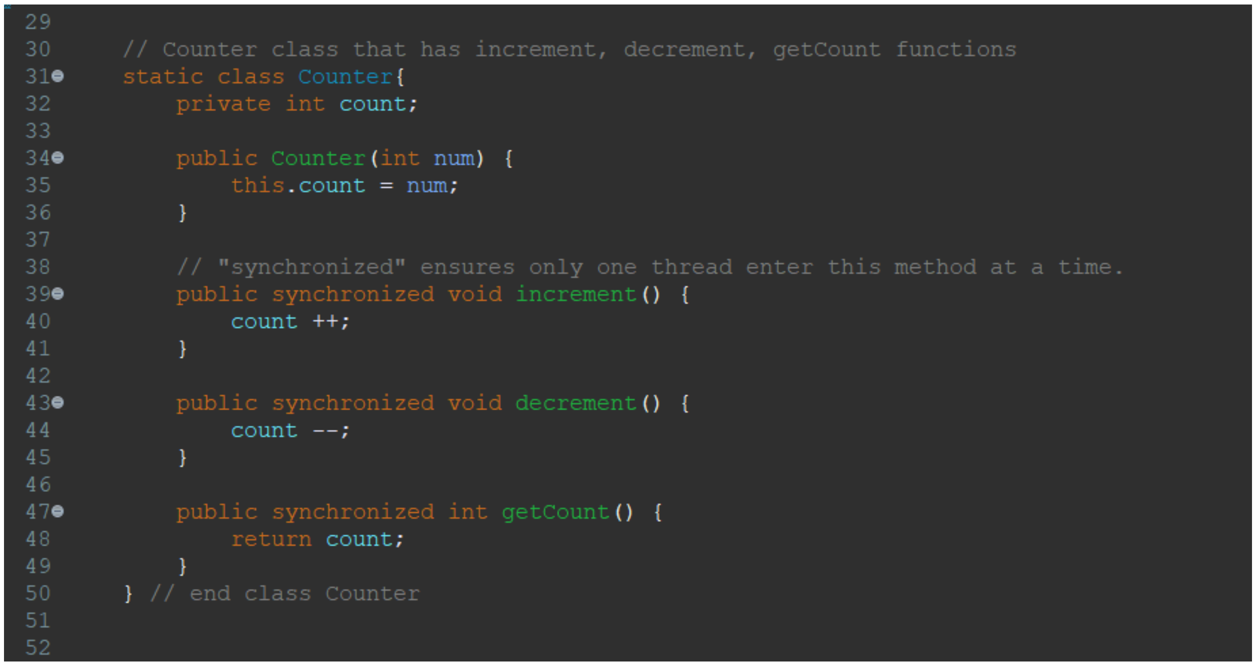
Third, we can use lambda expressions.

```
Thread t3 = new Thread(() -> {

          // define the thread’s behavior here

});
```

Among those three methods, I chose to build a class that inherits ‘Runnable’. Thus, I made two classes named Counter_up and Counter_down, and initiated them as Thread t1 and Thread t2.

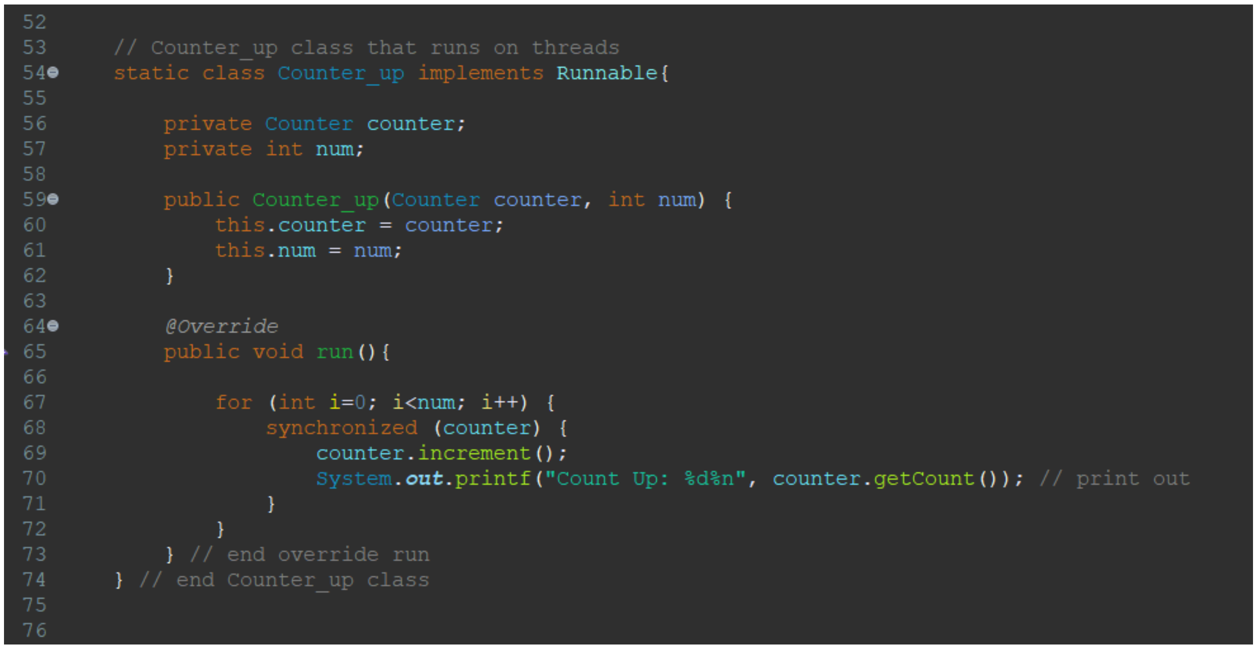
After initiating threads, I started and joined each of them one by one, making sure that t1 does its job before t2, meaning that counting up to 20 happens before counting down to 0. I also used the try catch method to control exceptions.

Then I defined Counter class, to define how the counter acts, as below. I used ‘synchronized’ to make sure these methods to be used in only one thread at a time.

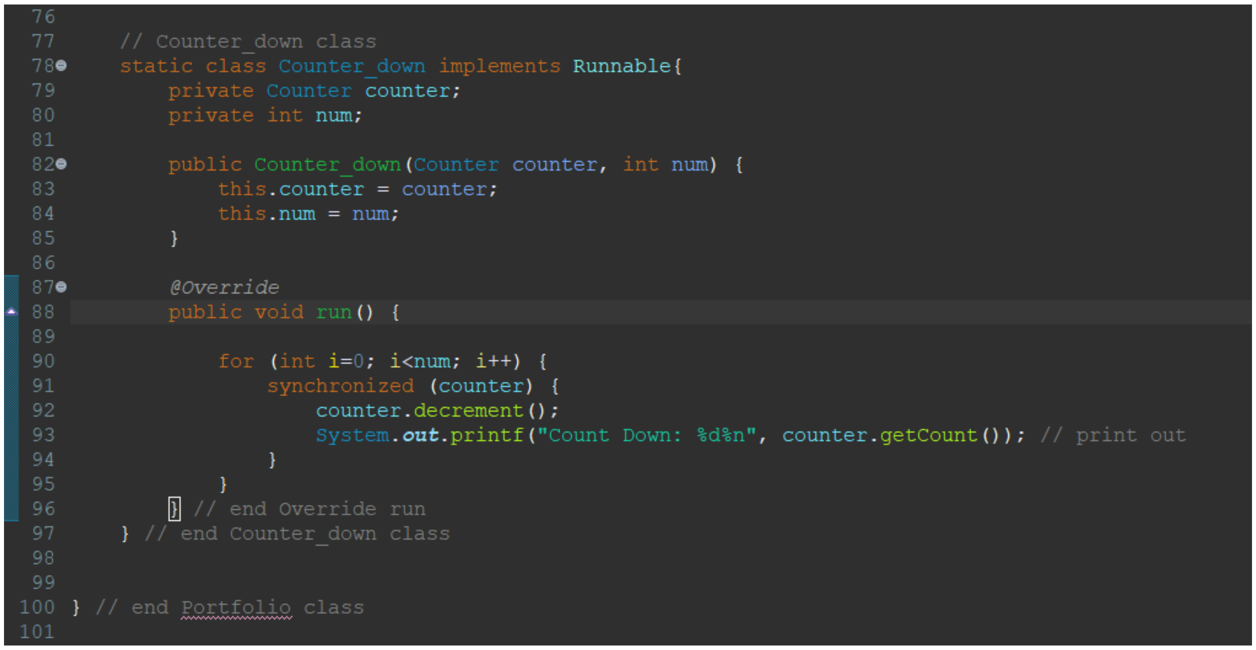


```java

    // Counter class that has increment, decrement, getCount functions
    static class Counter{
        private int count;

        public Counter(int num) {
            this.count = num;
        }

        // "synchronized" ensures only one thread enter this method at a time.
        public synchronized void increment() {
            count ++;
        }

        public synchronized void decrement() {
            count --;
        }

        public synchronized int getCount() {
            return count;
        }
    } // end class Counter


```

Lastly, I made these two Runnable methods called Counter_up and Counter_down. Those two use Counter objects, and for a given amount of time, it makes the counter increment or decrement by one. Code is like below.

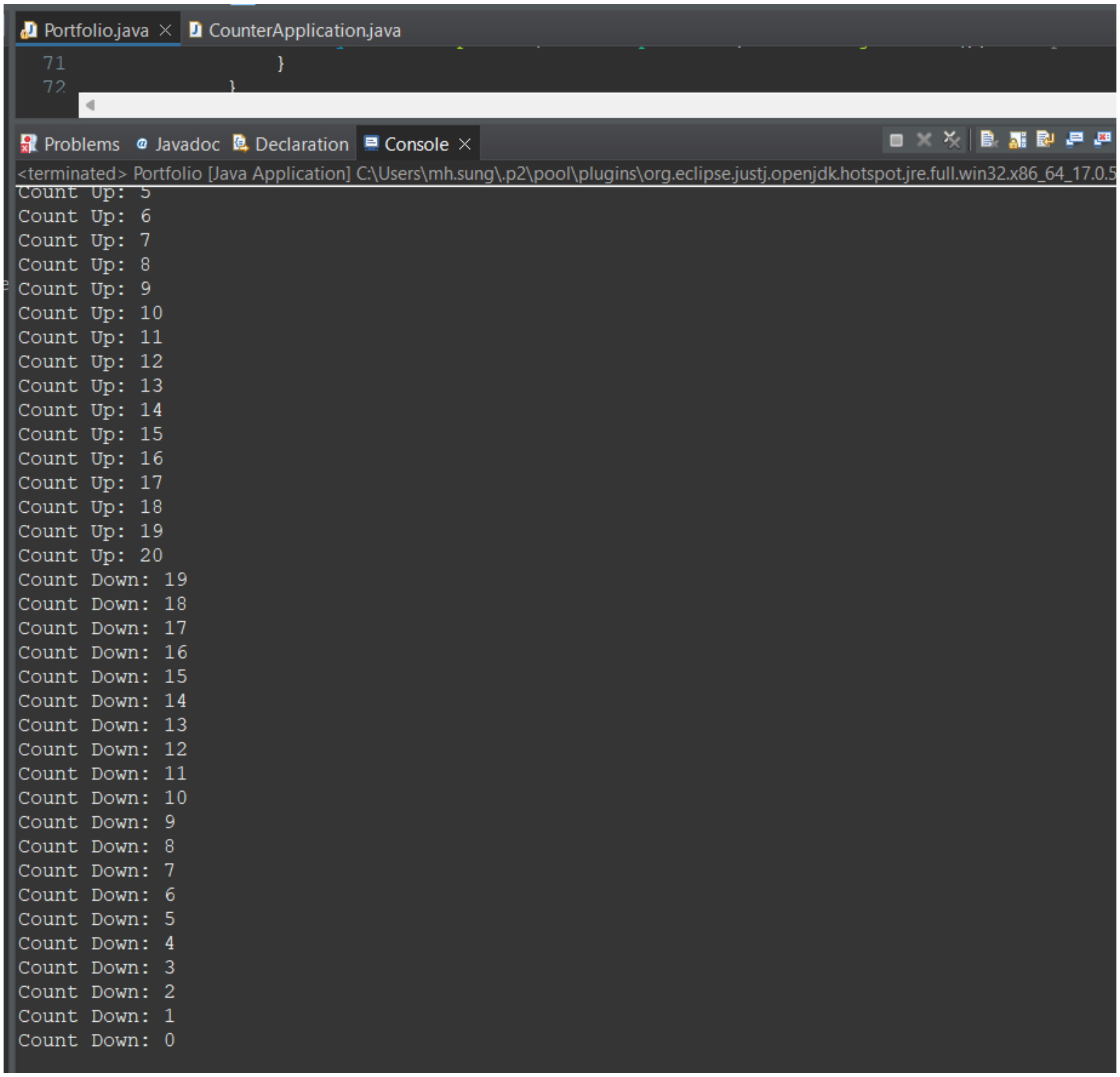


```java

    // Counter_up class that runs on threads
    static class Counter_up implements Runnable{

        private Counter counter;
        private int num;

        public Counter_up(Counter counter, int num) {
            this.counter = counter;
            this.num = num;
        }

        @Override
        public void run(){

            for (int i=0; i<num; i++) {
                synchronized (counter) {
                    counter.increment();
                    System.out.printf("Count Up: %d%n", counter.getCount()); // print out
                }
            }
        } // end override run
    } // end Counter_up class


```



```java

    // Counter_down class
    static class Counter_down implements Runnable{
        private Counter counter;
        private int num;

        public Counter_down(Counter counter, int num) {
            this.counter = counter;
            this.num = num;
        }

        @Override
        public void run() {

            for (int i=0; i<num; i++) {
                synchronized (counter) {
                    counter.decrement();
                    System.out.printf("Count Down: %d%n", counter.getCount()); // print out
                }
            }
        } // end Override run
    } // end Counter_down class


} // end Portfolio class

```

As a result, the application shows this result all the time, counting up first, and counting down later.



Portfolio.java CounterApplication.java

```
                    }
                }
```

Problems Javadoc Declaration Console

<terminated> Portfolio [Java Application] C:\Users\mh.sung\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.5

```
Count Up: 5
Count Up: 6
Count Up: 7
Count Up: 8
Count Up: 9
Count Up: 10
Count Up: 11
Count Up: 12
Count Up: 13
Count Up: 14
Count Up: 15
Count Up: 16
Count Up: 17
Count Up: 18
Count Up: 19
Count Up: 20
Count Down: 19
Count Down: 18
Count Down: 17
Count Down: 16
Count Down: 15
Count Down: 14
Count Down: 13
Count Down: 12
Count Down: 11
Count Down: 10
Count Down: 9
Count Down: 8
Count Down: 7
Count Down: 6
Count Down: 5
Count Down: 4
Count Down: 3
Count Down: 2
Count Down: 1
Count Down: 0
```

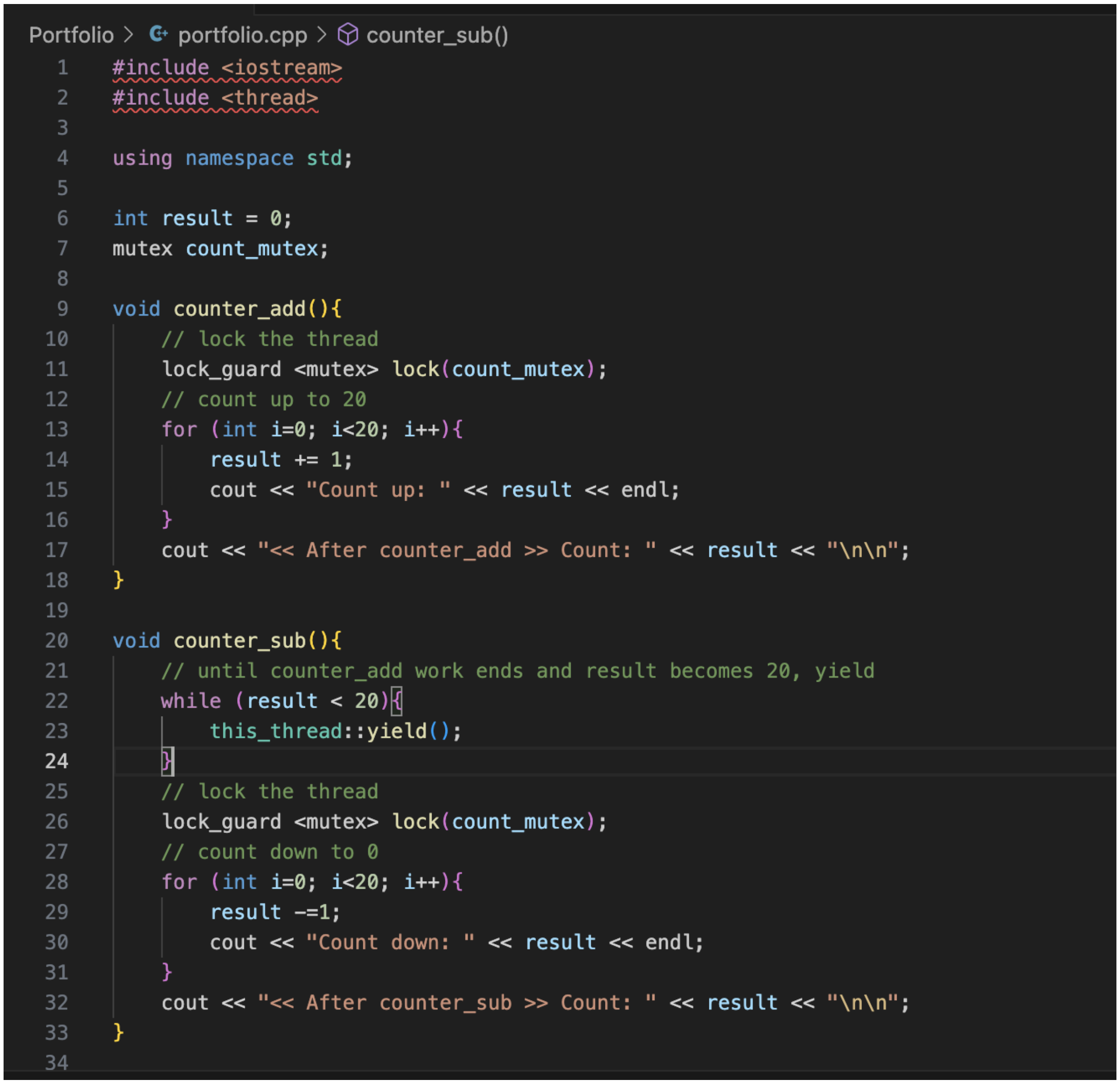
Part 2. Analysis

1) **Concurrency**: My java code uses synchronized methods and blocks to ensure that only one thread can access the ‘increment()’ and ‘decrement()’ methods of the ‘Counter’ class at a time. This ensures thread safety and prevents concurrent modifications to the shared Counter variable. The use of the ‘join’ method for t1 and t2 ensures that t1 finishes counting up before t2 starts counting down.

2) **Vulnerabilities**: The use of synchronization mechanisms helps prevent data races and concurrency issues. Also, it uses exception control to handle any exceptions or errors that might occur during the thread execution, which could be a potential vulnerability in more complex applications. This application does not utilize string data type, so there is no vulnerability related to the use of strings.

3) **Security**: This application uses ‘int’ for the counter variable, which is a basic and inherently thread-unsafe data type in nature. However, by using synchronized methods, the code mitigates the possible security issues, to protect access to the counter object.

Part 3. Comparison to C++ Code


Portfolio > portfolio.cpp > counter_sub()

```cpp
#include <iostream>
#include <thread>

using namespace std;

int result = 0;
mutex count_mutex;

void counter_add(){
    // lock the thread
    lock_guard <mutex> lock(count_mutex);
    // count up to 20
    for (int i=0; i<20; i++){
        result += 1;
        cout << "Count up: " << result << endl;
    }
    cout << "<< After counter_add >> Count: " << result << "\n\n";
}

void counter_sub(){
    // until counter_add work ends and result becomes 20, yield
    while (result < 20){
        this_thread::yield();
    }
    // lock the thread
    lock_guard <mutex> lock(count_mutex);
    // count down to 0
    for (int i=0; i<20; i++){
        result -=1;
        cout << "Count down: " << result << endl;
    }
    cout << "<< After counter_sub >> Count: " << result << "\n\n";
}

```

The above C++ code uses a ‘std::mutex’ to provide mutual exclusion for access to the shared counter. This ensures that only one thread can access the critical section of incrementing or decrementing the counter at a time. In comparison, Java code uses synchronization mechanisms. Both effectively prevent data races and concurrency issues during implementation.

During my research, I found out that ‘lock framework works like synchronized blocks except locks can be more sophisticated than Java’s synchronized blocks. (Geeks for geeks, 2023). It explains the reason is because ‘synchronization allows only one thread to access only one method at any given time, and it is a very expensive operation’. In that sense, I think C++’s mutex should be more similar to Java’s lock, making a better performance than synchronization.

My counter application uses only two threads, and one thread should be done before the other. Thus I do not think that using lock instead of synchronization in Java would make a significant performance difference in this specific given example. However, it should be worth remembering that lock and synchronization work differently, and that we need to differentiate the cases and situations in implementations.

In terms of security and vulnerability, both C++ and Java applications are safe and secure. The primary security threats in multithreaded programs are data races and concurrent access to shared resources. They are addressed by synchronization blocks and mutex locks in both applications. C++ program has one more feature of thread yield() to prevent Thread 2 happening before Thread 1 is done. I was able to manage it in Java differently, by starting Thread 2 after Thread 1 is already joined. That yield feature gave some disadvantage to the counter_sub() method in C++, which I will discuss more about soon.

They both did not use String data types, so there are no threats regarding the String data type. There could be overflow concerns regarding the usage of integers, however it can be easily handled by limiting the maximum bounds of the data. My C++ application did not allow the user to specify the number to which one wants to count up or down. It definitely makes the application safer and secure, however at the same time, it limits the user's free utilization of the application.

In Java, I made Counter_down() and Counter_up() methods to be able to take a number as its parameter by which the user wants to count. If a user does Counter_down(counter, 20), regardless of the current value of the counter, it subtracts 20 from the counter value. Thus, the user has more freedom to make choices, in execution order and in deciding the count number. If the Thread 1 was to count down, the result would have been counting down to 20, then counting up to 0.

In comparison, C++ application cannot do the above, because its count down method has a limiting condition of yield, that blocks the thread to be happening before the counter first reaches up to 20. It happened because I had to control the thread's execution order within the method, not outside of it. If I was able to control the execution order of Threads in the main() method, the actual count up and count down methods could be more free from Thread operations, possessing bigger possibilities to be used in more general cases.

Thus, I would conclude that this specific C++ code is more guarded in terms of outputting the same results all the time, however it does not mean that Java code is more unsafe than C++. The difference is from the code architecture, the design of abstraction, to be specific, to where I decided to give authorities or power of controlling the Thread execution orders, not from the genuine capabilities of both languages of C++ and Java.

During research, I found out that “the difference between C++ and Java in multithreading lies in the level of abstraction you have available for simplifying the writing of concurrent programs. … Java has long provided more tools and built-in functions for writing concurrent code. C++ does have a slight performance advantage here, though, due to it being closer to the hardware” (Shiotsu, 2021). I think this summarizes all well. Java has more applicable functions, as a more sophisticated language, while C++ has better performance in general, as a lower-level language.

<Reference>

(2023). Lock framework vs Thread synchronization in Java. Geeks for Geeks. https://www.geeksforgeeks.org/lock-framework-vs-thread-synchronization-in-java/

Y. Shiotsu. (2021). Java vs C++: Which language is right for your software project?. Upwork.com. https://www.upwork.com/resources/java-vs-c-which-language-is-right-for-your-software-project#:~:text=The%20difference%20between%20C%2B%2B%20and,addition%20of%20C%2B%2B11.