**Portfolio Project**

Mark Norem

Colorado State University Global

ITS450-1: Programming III

Instructor Reginald Haseltine

October 6th, 2024

## Analysis and Comparison of Java and C++

In my experience with developing both Java and C++ applications, I found concurrency to be a critical aspect that significantly impacts performance and the overall design of the application. Each language handles concurrency differently, which led to some interesting results when I implemented similar functionalities in both. Notably, the Java application took 21.462 seconds to complete, while the C++ application finished slightly faster at 21.404 seconds. These minor differences in execution time highlighted the trade-offs inherent in the choice of programming language.

**Performance Issues with Concurrency**

In the Java application, I relied on the Java Virtual Machine (JVM) for thread management, which provided built-in concurrency support. I appreciated how the JVM abstracted thread management, making it simpler to implement multi-threaded functionality. However, I also recognized that this abstraction comes with some performance overhead. The JVM requires additional resources to manage threads, which can introduce latency, particularly in CPU-bound tasks. I was aware of the potential pauses during garbage collection, where the JVM reclaims memory, which could slightly impact performance if not managed effectively.

In contrast, when developing the C++ application, I directly used the <thread> library, which allowed me to interact with the operating system threads more closely. This direct interaction resulted in better performance in this context, especially since there were no garbage collection pauses like in Java. I appreciated the control I had over thread behavior and resource management, but this flexibility also came with the responsibility of ensuring proper memory handling. The differences in how each language manages threads and memory were evident when I compared the execution times, though both applications performed similarly for the given tasks.

**Vulnerabilities Exhibited with the Use of Strings**

In my Java implementation, I used the immutability of strings to my advantage. I found that since Java strings cannot be modified after creation, they provided inherent thread safety. Multiple threads could safely read shared string objects without the risk of race conditions, simplifying my implementation. However, I did notice that frequent concatenation of strings could lead to performance issues because each modification creates a new string object, which increases memory allocation.

Conversely, in the C++ application, I worked with mutable strings using std::string. While this offered greater flexibility for string manipulation, it introduced potential vulnerabilities. I had to ensure proper synchronization when accessing shared string objects to avoid race conditions. Additionally, I recognized the risk of buffer overflows if I had opted to use raw character arrays instead of std::string. The manual memory management in C++ demanded more diligence from me to prevent concurrency-related issues, especially when sharing mutable data among threads.

**Security of the Data Types Exhibited**

In terms of security, I observed that Java provided a more robust environment for multi-threaded applications. The automatic memory management through garbage collection helped me avoid common security vulnerabilities like memory leaks or use-after-free errors. Additionally, the immutable data types in Java further ensured that data integrity was maintained when shared across multiple threads. I also appreciated the built-in thread-safe collections, which allowed me to focus more on application logic rather than managing the complexities of concurrent access.

On the other hand, the C++ application placed the onus of security on me as the developer. I needed to manage memory manually, which could lead to potential vulnerabilities such as memory leaks and dangling pointers if not handled carefully. The mutable nature of many C++ data types, including std::string, meant that I had to be vigilant about synchronization when multiple threads accessed shared data. I recognized that while C++ offered superior performance and control, it also increased the likelihood of errors that could compromise security.

**Comparison Between Java and C++ Implementations**

When comparing my experiences with the Java and C++ implementations, I found notable differences in handling concurrency, vulnerabilities, and security. The execution times of the two programs were quite close, with Java taking 21.462 seconds and C++ finishing at 21.404 seconds. This minor discrepancy highlighted the overhead introduced by the JVM in the Java version, though the difference was negligible for the tasks at hand.

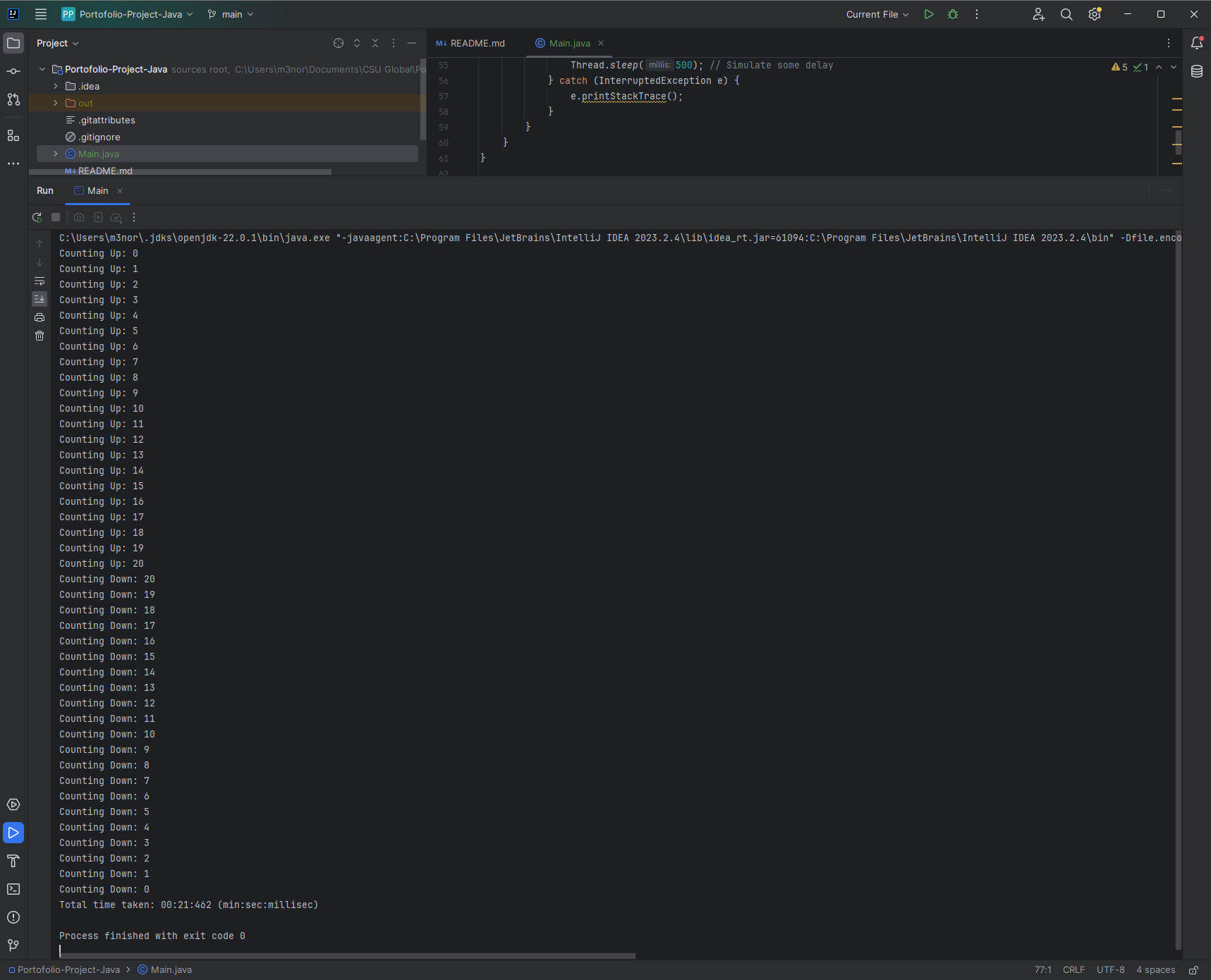
In terms of vulnerabilities, I appreciated Java's immutability and automatic synchronization mechanisms, which made it less prone to errors associated with concurrent access to strings. Meanwhile, C++ required more careful management of shared mutable data, with the potential for race conditions and memory-related vulnerabilities. The need for explicit synchronization in C++ demanded greater attention and discipline, which could lead to errors if overlooked.

Finally, from a security standpoint, I found Java to be more secure for multi-threaded applications due to its automatic memory management and thread-safe constructs. In contrast, the C++ application required me to carefully manage memory and synchronization, which increased the risk of errors and security vulnerabilities.

**Conclusion**

In conclusion, while both Java and C++ provided effective means to implement concurrency, my experience showed that Java is generally more secure and easier to use for multi-threaded applications. The automatic memory management and built-in thread safety features in Java allowed me to focus on application logic rather than the complexities of managing concurrent access. However, C++ remains a strong choice for performance-critical applications, where developers are willing to navigate the additional complexity and security risks.

## Code Execution Java:

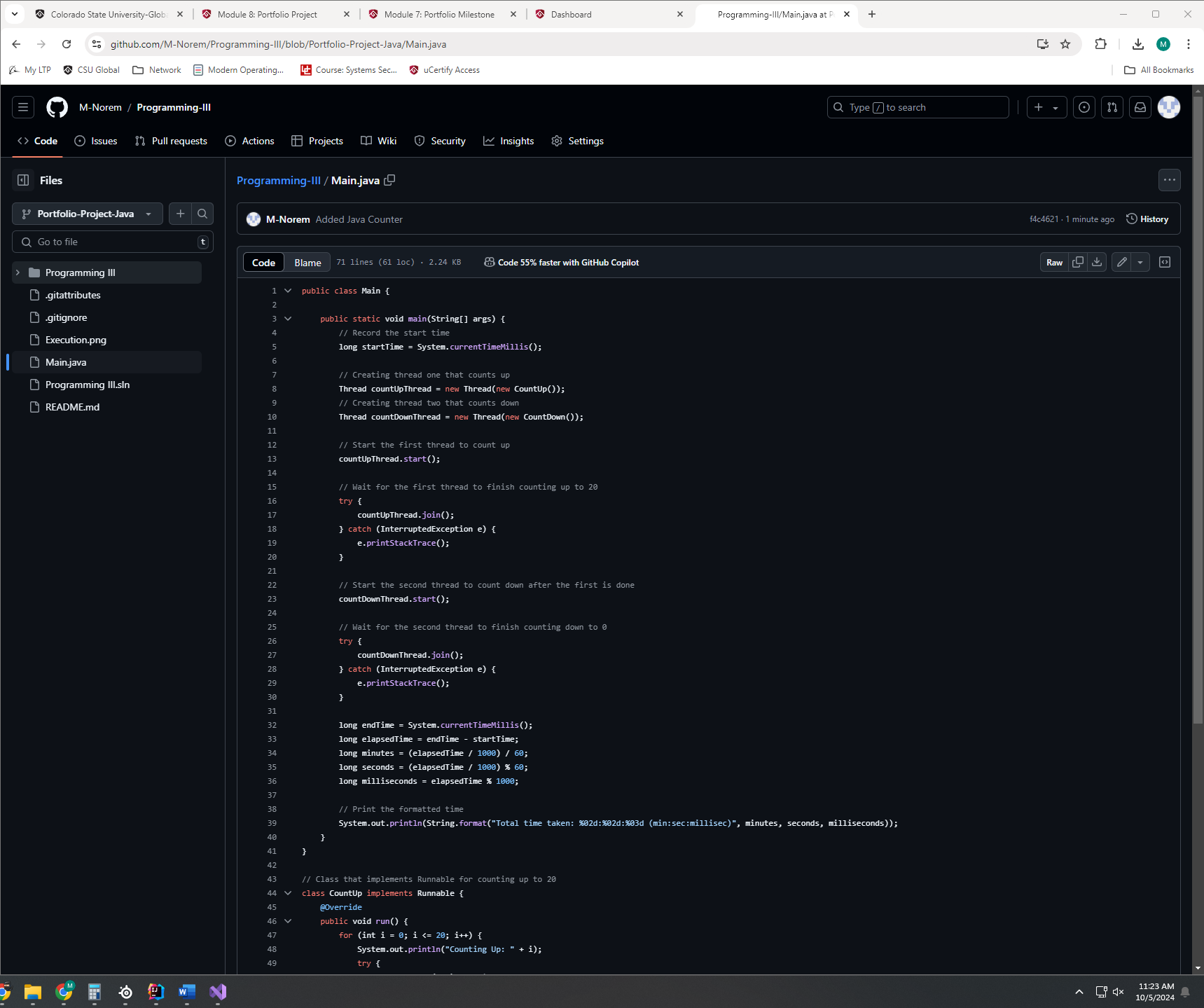


## Code Execution C++:

A screenshot of a computer program

Description automatically generated

## GitHub Repository (Java): <https://github.com/M-Norem/Programming-III/tree/Portfolio-Project>



## GitHub Repository (C++): <https://github.com/M-Norem/Programming-III/blob/Portfolio-Project/Main.cpp>

A screenshot of a computer

Description automatically generated

## Source Code (Java):

public class Main {

public static void main(String[] args) {

// Record the start time

long startTime = System.currentTimeMillis();

// Creating thread one that counts up

Thread countUpThread = new Thread(new CountUp());

// Creating thread two that counts down

Thread countDownThread = new Thread(new CountDown());

// Start the first thread to count up

countUpThread.start();

// Wait for the first thread to finish counting up to 20

try {

countUpThread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Start the second thread to count down after the first is done

countDownThread.start();

// Wait for the second thread to finish counting down to 0

try {

countDownThread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

long endTime = System.currentTimeMillis();

long elapsedTime = endTime - startTime;

long minutes = (elapsedTime / 1000) / 60;

long seconds = (elapsedTime / 1000) % 60;

long milliseconds = elapsedTime % 1000;

// Print the formatted time

System.out.println(String.format("Total time taken: %02d:%02d:%03d (min:sec:millisec)", minutes, seconds, milliseconds));

}

}

// Class that implements Runnable for counting up to 20

class CountUp implements Runnable {

@Override

public void run() {

for (int i = 0; i <= 20; i++) {

System.out.println("Counting Up: " + i);

try {

Thread.sleep(500); // Simulate some delay

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

// Class that implements Runnable for counting down from 20 to 0

class CountDown implements Runnable {

@Override

public void run() {

for (int i = 20; i >= 0; i--) {

System.out.println("Counting Down: " + i);

try {

Thread.sleep(500); // Simulate some delay

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

## Source Code (C++):

#include <iostream>

#include <thread>

#include <mutex>

#include <chrono>

#include <iomanip>

std::mutex mtx;

void countUp() {

for (int i = 0; i <= 20; ++i) {

std::lock\_guard<std::mutex> lock(mtx);

std::cout << "Count Up: " << i << std::endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

}

}

void countDown() {

for (int i = 20; i >= 0; --i) {

std::lock\_guard<std::mutex> lock(mtx);

std::cout << "Count Down: " << i << std::endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

}

}

int main() {

auto start = std::chrono::high\_resolution\_clock::now();

std::thread t1(countUp);

t1.join();

std::thread t2(countDown);

t2.join();

auto end = std::chrono::high\_resolution\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

long minutes = (duration / 1000) / 60;

long seconds = (duration / 1000) % 60;

long milliseconds = duration % 1000;

std::cout << "Total time taken: "

<< std::setw(2) << std::setfill('0') << minutes << ":"

<< std::setw(2) << std::setfill('0') << seconds << ":"

<< std::setw(3) << std::setfill('0') << milliseconds

<< " (min:sec:millisec)" << std::endl;

return 0;

}