**Portfolio Final Project**

Matthew Morrison

Colorado State University Global

CSC-450

Dr. Li

April 5, 2025

Here is the Java source code:

public class CounterApp {

private static volatile boolean countingComplete = false;

public static void main(String[] args) {

Thread counterUp = new Thread(new CounterUp());

Thread counterDown = new Thread(new CounterDown());

counterUp.start();

try {

counterUp.join(); // Wait for counterUp to finish

} catch (InterruptedException e) {

e.printStackTrace();

}

countingComplete = true; // Notify that the counting up is complete

counterDown.start(); // Start the countdown

}

static class CounterUp implements Runnable {

public void run() {

for (int i = 1; i <= 20; i++) {

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

try {

Thread.sleep(500); // Simulate work

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

static class CounterDown implements Runnable {

public void run() {

while (!countingComplete) {

// Wait for counting complete

}

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

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

try {

Thread.sleep(500); // Simulate work

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

Here is the [GitHub:](https://github.com/Irishmorrison/Portfolio-Final-/blob/main/README.md)

Here is the screenshot of the program executing:

A screenshot of a computer

AI-generated content may be incorrect.

**Performance Issues with Concurrency**

Concurrency can introduce several performance issues in the program including context switching, resource contention, and thread management overhead.

To begin, when multiple threads are active, the CPU may frequently switch between them, leading to overhead that can degrade performance. To explain, this is particularly true if threads are not designed to cooperate efficiently.

Secondly, if multiple threads are trying to access the same resources, it can lead to bottlenecks. For example, in our application, there is minimal contention since threads handle separate tasks, but in more complex applications, careful management is needed.

Thirdly, creating and managing threads incurs additional overhead. This overhead can become significant if the threads are short-lived or if many threads are created.

**Vulnerabilities Exhibited with use of Strings**

In Java, strings are immutable, which provides some level of security against certain vulnerabilities related to string manipulation.

To start, since strings cannot be altered after creation, the risk of injection attacks is minimized. However; care must still be taken when strings are concatenated or manipulated.

Secondly, if strings are not handled properly, especially in concurrent environments, it can lead to memory leaks. To explain, if threads hold onto references to strings longer than necessary, it can prevent garbage collection.

**Security of the Data Types Exhibited**

In this application, basic data types are used, which have following security implications.

To begin, the use of the volatile keyword for the counting Complete Boolean ensures that changes made by one thread are visible to other threads immediately. This prevents inconsistent views of the variables across threads.

Next, using primitive data types minimizes the overhead and security risks associated with objects, such as null references or mutable state.

**Comparison with C++ Implementation Performance Implementations**

In Java the garbage collector handles memory management, which can impact performance based on how frequently it runs. To explain, the thread management is built into the Java Virtual Machine (JVM), allowing for easier thread connection but potentially less control over thread scheduling.

In C++ programmers must manage memory explicitly, which can lead to better performance in some cases due to reduced overhead but increased complexity and risk of memory leaks.

Next, C++ allows for lower-level thread management via libraires like <thread> but requires more careful handling of concurrency primitives.

**Security Vulnerabilities**

Lastly, Java’s inherent memory management and type safety provides a layer of security against certain vulnerabilities compared to C++.

To begin, Java’s strong type system prevents many common vulnerabilities related to type casting and buffer overflow issues prevalent in C++

Secondly, Java’s automatic garbage collection and lack of pointers reduce the risk of dangling pointers and memory corruption compared to C++.

**Conclusion**

In conclusion, while both Java and C++ have their strengths in managing concurrency, Java’s built-in security features and automatic memory management make it less vulnerable to certain security threats. However, C++ can yield better performance with careful management and optimization. The choice between Java and C++ ultimately depends on the specific requirements of the application and the expertise of the development team.

**Reference**

Chapter 1 in [SEI CERT C++ coding standard: Rules for developing safe, reliable, and secure systems in C++](https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf)

Chapters 2 & 3 in [Introduction to programming using Java](http://math.hws.edu/javanotes/)

Chapters 4 & 5 in [*Introduction to programming using Java*](http://math.hws.edu/javanotes/)

Chapter 6 in [*SEI CERT C++ coding standard: Rules for developing safe, reliable, and secure systems in C++*](https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf)

Chapters 7 & 8 in [*Introduction to programming using Java*](http://math.hws.edu/javanotes/)

Chapter 9 in [SEI CERT C++ coding standard: Rules for developing safe, reliable, and secure systems in C++](https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf)

Chapters 8 & 11 in [*SEI CERT C++ coding standard: Rules for developing safe, reliable, and secure systems in C++*](https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf)