**Critical Thinking Week 8**

Tomas Toledo

Colorado State University Global

CSC450 Programming III

Professor George Bindu

3/8/2025

## Portfolio Project

Concurrency in this application allows both threads to execute simultaneously, improving efficiency by utilizing CPU resources more effectively. Multi-core processors can execute CounterUp and CounterDown in parallel, reducing overall execution time. However, since both threads are independent, the order of execution is unpredictable due to Java’s thread scheduling. The frequent context switching between threads can introduce some overhead, but given the simplicity of this program in particular, the performance impact remains minimal.

The use of Thread.sleep(100) introduces an artificial delay, simulating real-world processing and allowing both threads to have a chance to execute without one completely dominating the CPU. However, because sleep times are not exact and depend on system scheduling, the sequence of the printed output may vary between runs. While the current implementation works well for just two threads, scaling the program to handle multiple concurrent operations would require more advanced thread management techniques, like ExecutorService. Additionally, if strict sequencing were required. mechanisms such as locks or semaphores could be introduced to enforce controlled execution.

String manipulation in Java can introduce performance and security and vulnerabilities if not handled properly. Strings are immutable, meaning every time a new value is assigned to a string, a new object is created , potentially leading to memory inefficiencies. To address this, the updated implementation replaces String concatenation with StringBuilder, which is mutable and avoids unnecessary object creation. This change reduces the memory overhead and improves execution efficiency, especially in a loop that performs frequent modifications.

From a security perspective, improper string handling can sometimes lead to information leakage. In this case, the use of Java’s Logger instead of System.out.println() enhances security by providing better control over how log messages are stored and displayed. Each thread maintains its own instance of StringBuilder, ensuring there is no data corruption or race condition between them. Although StringBuilder is not thread-safe, it is safe to use in this case since each instance is confined to a single thread.

This application relies on primitive int variables to track the count for both threads, this choice was made with consideration of the relatively small size of the range of values, being only 0-20. Had the ranges been larger and the counter needed to handle larger values, using long or BigInteger would have been a more suitable choice. Since both threads maintain their separate counters, there is no risk of data races or corruption. If a shared counter were introduced, synchronization techniques such as synchronized methods or atomic variables like AtomicInteger would be necessary to ensure thread safety.

The implementation also improves security by replacing direct console output with a logging framework. Unlike System.out.println(), which exposes all output directly to the console, Logger allows for better log management, guaranteeing that sensitive data is not exposed by accident. Logs can be stored securely, formatted appropriately, and filtered as needed, reducing the risk of data leaks. Overall, by avoiding shared variables and using efficient data handling techniques, the application maintains both performance and security while demonstrating key concurrency concepts learned throughout this course.

## Comparison between Java and C++

In the Java program, thread management is handled by the Java Virtual Machine (JVM), which abstracts system-level threading details. The Java implementation uses the Thread class, with Thread.sleep(100) to introduce artificial delays and allow both threads to execute without one dominating CPU usage. However, Java thread scheduling depends on the JVM and underlying operating system, making execution timing less predictable.

C++ on the other hand, uses the standard library’s std::thread to create and manage threads. The C++ implementation uses std::mutex to synchronize console output, ensuring proper sequencing and preventing interleaved prints. Unlike Java, C++ threads operate at a lower level, closer to the hardware, leading to lower overhead and more precise control over execution.

Java applications run within the JVM, which introduces some overhead due to garbage collection and runtime management. The JVM dynamically allocates and deallocates memory, which can lead to latency spikes, especially if garbage collection occurs during execution. C++ directly manages memory without a virtual machine, leading to lower overhead and faster execution. The C++ implementation uses stack-allocated primitive types (int), ensuring minimal consumption. Since C++ does not have automatic garbage collection, it avoids the unpredictable delays involved with Java's memory management.

In Java, thread safety relies on features such as synchronized methods or explicit locks (e.g., ReentrantLock). The Java implementation avoids race conditions by keeping separate counters for each thread and avoiding shared data.

The C++ implementation explicitly manages synchronization using std::mutex. The std::lock\_guard<std::mutex> ensures mutual exclusion when writing to std::cout, preventing race conditions in console output. This approach provides deterministic control over synchronization, potentially leading to better performance in multi-threaded environments.

Java has built-in memory safety mechanisms, such as automatic bounds checking and garbage collection, reducing risks like buffer overflows and dangling pointers. The Java implementation only deals with primitive int variables, which are inherently safe from memory corruption. C++ offers greater control over memory but requires manual management, increasing the risk of memory-related vulnerabilities. While the provided C++ implementation is safe (since it only uses stack allocated primitives), in more complex applications, improper memory management could lead to issues such as buffer overflows, use-after-free errors, or memory leaks.

Java’s String is immutable and managed by the JVM, preventing unintended modifications. The Java implementation uses StringBuilder, which is more efficient for concatenation while still being safe from memory corruption. In C++, string handling can be a security risk if char\* or manual memory allocation (new/delete) is used. However, the Java implementation does not use strings beyond std::cout, which is safe. If dynamic memory allocation were involved, additional precautions like std::string or smart pointers would be necessary to prevent vulnerabilities.

In conclusion, Java’s vulnerabilities lie in the JVM’s garbage collector, the uncertainty of thread scheduling, and string immutability overhead, while C++’s vulnerabilities lie in manual memory management risks, mutex overhead, and the lack of automatic bounds checking.   
The Java implementation benefits from automatic memory management and built-in security features, making it more resilient against memory-related vulnerabilities. However, the JVM introduces performance overhead and unpredictability in thread execution. The C++ implementation offers lower-level control, resulting in better performance but requiring careful memory and thread management to avoid security risks.

For applications where performance is critical, C++ is the better choice due to its direct access to system resources and efficient threading. However, for applications where security and memory safety are a priority, Java provides a more robust and safer environment with less risk of manual memory management errors. As with most decisions in computer science, the decision is heavily dependent on the requirements of the program, and what it should accomplish. The choice between Java and C++ depends on the specific use case and security requirements.

**Source Code:**

/\*\*

\* This Java application demonstrates concurrency by utilizing two threads.

\* One thread counts up to 20, and another counts down from 20 to 0.

\* Both threads run concurrently without synchronization.

\*/

import java.util.logging.Logger;

public class Concurrency {

private static final Logger LOGGER = Logger.getLogger(Concurrency.class.getName());

public static *void* main(String[] *args*) {

// Create instances of both counter threads

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

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

// Start both threads simultaneously

counterUp.start();

counterDown.start();

}

}

/\*\*

\* CounterUp implements Runnable to count from 0 to 20.

\*/

class CounterUp implements Runnable {

private static final Logger LOGGER = Logger.getLogger(CounterUp.class.getName());

@*Override*

public *void* run() {

StringBuilder output = new StringBuilder();

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

output.setLength(0); // Reset buffer

output.append("Counter Up: ").append(i);

LOGGER.info(output.toString());

try {

Thread.sleep(100); // Simulate processing delay

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

LOGGER.severe("CounterUp interrupted: " + e.getMessage());

}

}

}

}

/\*\*

\* CounterDown implements Runnable to count from 20 to 0.

\*/

class CounterDown implements Runnable {

private static final Logger LOGGER = Logger.getLogger(CounterDown.class.getName());

@*Override*

public *void* run() {

StringBuilder output = new StringBuilder();

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

output.setLength(0); // Reset buffer

output.append("Counter Down: ").append(i);

LOGGER.info(output.toString());

try {

Thread.sleep(100); // Simulate processing delay

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

LOGGER.severe("CounterDown interrupted: " + e.getMessage());

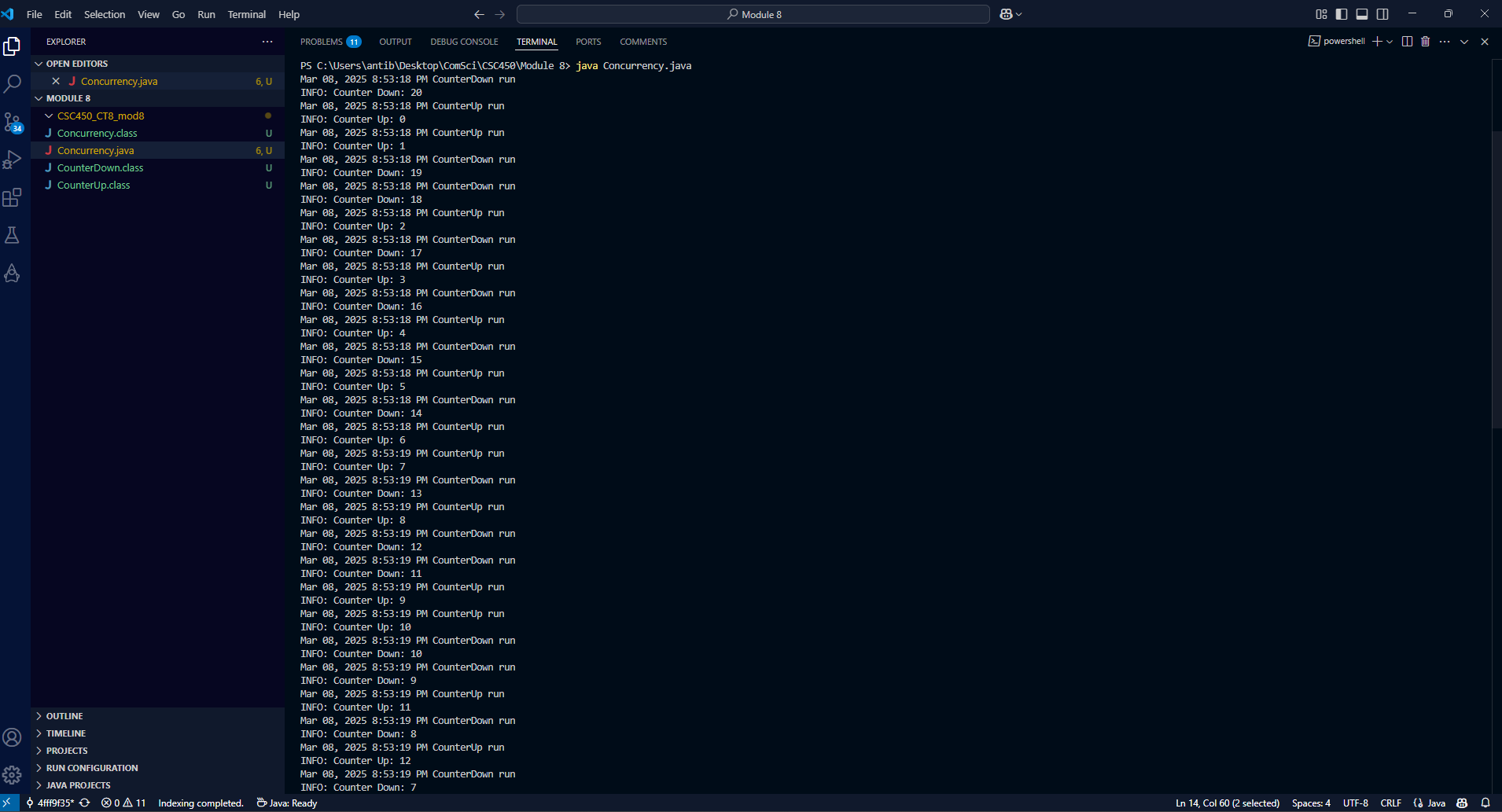
}

}

}

}

**Screenshots:**



**References**

Jain, S. (2024, May 16). *Concurrency in C++*. GeeksforGeeks. Retrieved March 8, 2025, from https://www.geeksforgeeks.org/cpp-concurrency/

*Java Concurrency: Master the Art of Multithreading*. (n.d.). BairesDev. Retrieved March 8, 2025, from https://www.bairesdev.com/blog/java-concurrency/