**Java Concurrency: Managing Sequential Threads**

Sarah Cole

Colorado State University Global

Course Code: CSC 450

Jack Li

March 30 , 2025

**Java Concurrency: Managing Sequential Threads**

This project explores the concept of concurrency in Java by implementing a multithreaded application that demonstrates sequential thread execution. The program creates two threads—one that counts upward from 0 to 20 and another that counts downward from 20 to 0. The second thread begins only after the first completes, using Java’s ‘Thread’ and ‘join()’ methods to coordinate execution.

Concurrency enables applications to perform multiple tasks simultaneously, improving responsiveness and resource utilization (2023). However, it also introduces risks such as race conditions, deadlocks, and performance degradation if not properly managed (Kurve, 2024). This implementation emphasizes safe and structured thread control while exploring how thread synchronization affects overall program flow.

In addition to functionality, this project evaluates critical software design considerations such as string handling and data type security in concurrent environments. The Java application will be compared to a parallel implementation written in C++, with attention to each language’s approach to thread management, security vulnerabilities, and execution efficiency. Through this comparison, the project aims to identify which language offers stronger performance and reliability for concurrent programming scenarios.

**Implementation of the Program**

This program was implemented in Java using a simple concurrency model involving two threads. The ‘ConcurrentThreadRunner’ class serves as the main entry point, creating and managing two worker threads: ‘CountUpTask’ and ‘CountDownTask’. The first thread counts from 0 to 20, and upon completion, the second thread counts down from 20 to 0. Java’s Thread class and the ‘Runnable’ interface were used to define thread behavior, and the ‘join()’ method ensured proper synchronization by forcing the second thread to wait until the first completed.

Each thread uses a basic ‘for’ loop along with ‘Thread.sleep()’ to simulate workload and create visible output delays. This project demonstrates sequential thread execution without shared memory or complex synchronization, making it ideal for showcasing fundamental concurrency control using Java’s built-in threading mechanisms.

**Pseudocode**

A screenshot of a computer program

AI-generated content may be incorrect.

**Source Code**

A screen shot of a computer program

AI-generated content may be incorrect.

**Outcome**

A screenshot of a computer

AI-generated content may be incorrect.

**GitHub URL**

https://github.com/skc1339/CSC450-Mod-8-Portfolio-Milestone/tree/main

**Program Analysis**

This section evaluates the concurrency-related design decisions and implementation outcomes of the Java program. While the application is relatively simple, it demonstrates core concurrency principles through thread coordination and control flow. The analysis focuses on three key areas: performance issues that can arise from using threads, potential string-related vulnerabilities, and the security implications of the data types used.

### **Performance Issues with Concurrency**

Concurrency introduces the potential for improved efficiency and responsiveness in applications, but it also presents risks when not managed properly. In this program, Java threads are used in a controlled manner to ensure sequential execution. The ‘join()’ method plays a key role in performance control by blocking the main thread until the ‘CountUpTask’ completes. This prevents unnecessary context switching and avoids potential thread overlap, which could otherwise degrade performance.

While the concurrency here is straightforward, more complex multithreaded applications can suffer from race conditions, deadlocks, or excessive thread creation—all of which negatively impact performance. Additionally, Java threads are relatively lightweight but still involve overhead due to stack space allocation and scheduling by the JVM. In long-running or highly concurrent applications, this overhead can become significant if thread creation is not properly managed (2006).

Modern Java (starting with version 19) also introduces virtual threads (Project Loom) to reduce the performance cost of thread management by decoupling thread usage from OS-level threads. Although not used in this project, these alternatives are relevant when scaling concurrent applications with lower memory usage and faster context switching (2023).

### **Vulnerabilities Exhibited with Use of Strings**

Java strings are immutable, which generally contributes to safety and predictability, particularly in multithreaded applications. However, this immutability also introduces a subtle vulnerability when handling sensitive information. For instance, if passwords or personal data are stored in a ‘String’, they cannot be explicitly erased from memory since they remain in the heap until garbage collection occurs. This opens the possibility of exposing sensitive data if memory is compromised (Weligalla, 2024).

Although this particular project does not handle user input or sensitive strings, awareness of this behavior is important for future applications. The recommended practice is to use character arrays (‘char[]’) when working with confidential information, as they can be manually cleared from memory. Furthermore, frequent string manipulation in a concurrent context (e.g., using string concatenation inside loops or threads) may generate a large number of intermediate string objects, potentially increasing garbage collection frequency and reducing performance (Kurve, 2024).

### **Security of the Data Types Exhibited**

The data types used in this program are local ‘int’ variables scoped within each thread’s loop. Because there is no shared memory or global state, this implementation is inherently thread-safe. Each thread manages its own loop variable without requiring synchronization mechanisms like locks or atomic classes. This design avoids the complexity of race conditions or memory visibility issues that arise when threads share mutable state.

From a security perspective, the use of primitive data types in isolated thread contexts is one of the safest approaches to concurrency. If the application were extended to involve shared resources (such as counters, logs, or buffers), it would be necessary to introduce thread-safe data structures like ‘AtomicInteger’ or ‘ConcurrentHashMap’ from the ‘java.util.concurrent’ package to preserve data integrity across threads (2006).

Additionally, exception handling is implemented to catch ‘InterruptedException’ within each thread. This ensures that if a thread is interrupted during sleep, it fails gracefully rather than terminating the program unexpectedly. Logging errors to the console with descriptive messages enhances the application’s resilience and aids debugging.

**Java vs. C++: Concurrency Performance Comparison**

Both Java and C++ are powerful languages capable of handling concurrent programming, but they differ significantly in how they implement concurrency, manage resources, and address security vulnerabilities. This section provides a detailed comparison between the two languages based on their performance characteristics and vulnerability to security threats, particularly within the context of thread management and data handling.

From a performance standpoint, Java provides a more accessible and developer-friendly concurrency model. Java's built-in ‘Thread’ class, ‘Runnable’ interface, and newer concurrency utilities such as ‘Executors’ and ‘CompletableFuture’ offer high-level abstractions that reduce the cognitive load on developers. The use of ‘join()’ in this project ensures that one thread completes before the next begins, preventing simultaneous access to resources and avoiding race conditions. Java also benefits from Just-In-Time (JIT) compilation within the Java Virtual Machine (JVM), which can optimize long-running processes and improve execution speed over time. However, Java's reliance on the JVM also introduces overhead, especially when threads are short-lived or memory is heavily used. Additionally, garbage collection can pause threads temporarily, which, while usually minor, may introduce latency in performance-sensitive applications (2023).

C++, in contrast, offers finer control over hardware and system resources, which can lead to better raw performance when concurrency is implemented carefully. Using ‘std::thread’, developers can spawn threads efficiently, and with the help of synchronization primitives like ‘std::mutex’ or ‘std::condition\_variable’, precise control over execution timing and resource sharing is possible. Unlike Java, C++ does not abstract away memory management, which allows for highly optimized programs—but at the cost of increased complexity. A small oversight in thread synchronization or memory management can result in undefined behavior, including deadlocks or segmentation faults. These issues are often difficult to detect and resolve, especially in large-scale or multithreaded applications (2012).

When evaluating the security aspects of concurrency in both languages, Java holds a clear advantage due to its design. Java enforces strict memory safety through automatic bounds checking, strong typing, and runtime exception handling. For example, attempting to access an invalid array index in Java will result in an ‘ArrayIndexOutOfBoundsException’, preserving program integrity. In C++, however, such behavior might lead to a buffer overflow—potentially corrupting memory or allowing arbitrary code execution. Similarly, Java’s exception handling model ensures that errors like null references are caught and reported, whereas C++ may allow the program to crash if a ‘nullptr’ is dereferenced without appropriate checks.

Another critical security concern lies in how each language handles strings and memory. Java strings are immutable and managed by the JVM, which prevents developers from inadvertently overwriting or corrupting memory through string operations. However, this immutability can pose a risk when sensitive data—such as passwords—is stored in a ‘String’ object, as it cannot be cleared from memory manually. For this reason, Java security best practices recommend using ‘char[]’ arrays for storing sensitive data, as these can be explicitly wiped (Weligalla, 2024). In contrast, C++ gives developers full control over memory and allows manual overwriting of character buffers, but this comes at the cost of increased responsibility and risk. Improper memory handling in C++ has led to well-known vulnerabilities, including buffer overflows and data leakage (2003).

In multithreaded applications, shared mutable state introduces additional risk. Java provides several built-in tools to promote thread safety, such as ‘synchronized’ blocks, atomic variables like ‘AtomicInteger’, and collections from the ‘java.util.concurrent’ package. These abstractions reduce the chance of race conditions and deadlocks and are part of the Java memory model that ensures predictable behavior across threads (2006). In contrast, C++ developers must manually manage thread synchronization using ‘std::mutex, std::unique\_lock’, and other lower-level primitives. While this grants more flexibility, it also increases the likelihood of subtle bugs if mutexes are misused or forgotten (Sutter & Alexandrescu, 2005).

Exception handling further distinguishes the two languages in terms of reliability. Java enforces checked exceptions, requiring developers to either catch or declare them. This design encourages explicit error handling and improves code clarity, particularly in large applications. C++ also supports exception handling, but it is optional and often avoided in favor of return codes due to performance concerns and undefined behavior in certain compiler configurations (Meyers, 2005). Additionally, Java’s exception stack traces provide detailed information useful for debugging, while unhandled exceptions in C++ can lead to program crashes and harder-to-trace errors.

Type safety and casting mechanisms also influence the security profile of each language. Java promotes strict type checking at both compile-time and runtime, reducing the risk of type confusion and runtime errors. Generics help enforce type safety in collections and reduce the use of unsafe casts. On the other hand, C++ permits powerful but risky casting operations like ‘reinterpret\_cast’, which can result in undefined behavior or memory corruption when used incorrectly (Sutter & Alexandrescu, 2005). In multithreaded environments, unsafe casting can compound the difficulty of tracking memory consistency issues.

In real-world applications, Java is widely used in enterprise systems, web applications, and Android development, largely due to its balance between performance and safety. Its concurrency model and automatic memory management make it particularly well-suited for scalable server-side applications where uptime and security are critical (2006). C++, meanwhile, dominates in areas where direct hardware interaction, real-time performance, and memory efficiency are essential—such as in game engines, robotics, and embedded systems (2012).

In the context of this project, both implementations successfully execute concurrent tasks in a controlled and predictable sequence. However, Java’s design choices—such as managed memory, built-in thread safety tools, and enforced exception handling—make it a safer language by default. C++ can offer superior performance, but this advantage comes with significant trade-offs in terms of complexity and vulnerability potential. As such, for projects where security, maintainability, and rapid development are priorities, Java is often the preferred choice. For applications demanding the highest levels of performance and hardware efficiency, C++ remains essential—provided that developers apply rigorous safeguards throughout the codebase.

**References**

Goetz, B. (2006). *Java Concurrency in Practice*. Addison-Wesley Professional.

Howard, M., & LeBlanc, D. (2003). *Writing Secure Code* (2nd ed.). Microsoft Press.

Kurve, S. (2024, October 30). *Best Practices and Pitfalls in Java Exception Handling*. DEV.

<https://dev.to/saurabhkurve/best-practices-and-pitfalls-in-java-exception-handling-37dk>

Meyers, S. (2005). *Effective C++* (3rd ed.). Addison-Wesley.

Oracle. (2023). *Concurrency (The Java™ Tutorials > Essential Classes)*.

<https://docs.oracle.com/javase/tutorial/essential/concurrency/>

Sutter, H., & Alexandrescu, A. (2005). *C++ Coding Standards: 101 Rules, Guidelines, and Best*

*Practices*. Addison-Wesley.

Weligalla, N. (2024, January 31). *Creating Custom Exceptions in Java*. Medium.

<https://medium.com/@nweligalla/creating-custom-exceptions-in-java-ea77a61fcaf4>

Williams, A. (2012, February). *C++ concurrency in action: Practical multithreading*. Manning

Publications. https://www.manning.com/books/c-plus-plus-concurrency-in-action

Kurve, S. (2024, October 30). *Best Practices and Pitfalls in Java Exception Handling*. DEV.

<https://dev.to/saurabhkurve/best-practices-and-pitfalls-in-java-exception-handling-37dk>