CSC450 Portfolio Project C++ vs Java

Dennis Weddig

Colorado State University Global

CSC450: Programming III

Dr. Jack Li

10/5/2025 11:59pm

csc450 Portfolio Project C++ vs Java

Per the assignment, this first section is the Part II of the portfolio project, which is the Java implementation of concurrently counting up to, and then down from, 20 with two different threads, the countdown task dependent on the count up task. The code is below and includes within it the discussion points about best practices, security and performance issues addressed within the code.

All my code, notes, references and research can be found at https://github.com/denniswed/csc450 .

```
package Portfolio.Module8;

/**
* CSC450 Portfolio Module 8 - Concurrency Counters Application
*
* ASSIGNMENT REQUIREMENTS:
* - Create two threads that act as counters
* - Thread 1: Count up to 20
* - Thread 2: Count down from 20 to 0 (after Thread 1 completes)
*
* ANALYSIS TOPICS:
* 1. Performance issues with concurrency
* 2. Vulnerabilities with use of strings
* 3. Security of data types
*
* IMPROVEMENTS IMPLEMENTED:
* - Thread-safe synchronization using CountDownLatch
* - StringBuilder for efficient string concatenation
* - Proper resource management with ExecutorService
* - Exception handling with interrupt status preservation
* - Comprehensive logging and error handling
* - Security-conscious design patterns
*
* @author CSC450 Student
* @version 2.0 - Enhanced with best practices and security improvements
*/
```

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
* Demonstrates proper concurrent counting with thread coordination.
*
* PERFORMANCE OPTIMIZATIONS:
* - Uses StringBuilder to minimize string allocation overhead
* - CountDownLatch for efficient thread coordination (no busy-waiting)
* - Fixed thread pool to prevent thread creation overhead
* - Atomic operations where needed for thread safety
*
* SECURITY CONSIDERATIONS:
* - Immutable counter bounds (final constants)
* - No data exposure through shared mutable state
* - Proper exception handling prevents information leakage
* - Thread interruption handled securely
*/
public class ConcurrencyCounters {

// SECURITY: Final constants prevent modification - immutable configuration
private static final int MAX_COUNT = 20;
private static final int THREAD_POOL_SIZE = 2;
private static final int SHUTDOWN_TIMEOUT_SECONDS = 5;
private static final int STRING_BUILDER_CAPACITY = 128;

// BEST PRACTICE: Use proper logging instead of System.out for production
private static final Logger logger = Logger.getLogger(ConcurrencyCounters.class.getName());

// PERFORMANCE: Track execution metrics
private static final AtomicInteger threadExecutionCount = new AtomicInteger(0);

public static void main(String[] args) {
logger.info("=== Concurrency Counters Application Started ===");
logger.info("Configuration: MAX_COUNT=" + MAX_COUNT +
", THREAD_POOL_SIZE=" + THREAD_POOL_SIZE);

long startTime = System.nanoTime();
```

```
try {
runConcurrencyDemo();

long duration = System.nanoTime() - startTime;
logger.info(String.format("=== Application Completed Successfully in %.2f ms ===",
duration / 1_000_000.0));

} catch (Exception e) {
logger.log(Level.SEVERE, "Application failed with error", e);
System.exit(1);
}
}

/**
* Executes the concurrent counting demonstration.
*
* CONCURRENCY PATTERN:
* Uses CountDownLatch as a synchronization barrier to ensure:
* 1. Thread 1 completes counting up before Thread 2 starts counting down
* 2. No race conditions or data corruption
* 3. Efficient signaling without polling or sleep
*
* PERFORMANCE ANALYSIS:
* - ExecutorService reuses threads (no creation overhead per task)
* - CountDownLatch uses lock-free operations when possible
* - StringBuilder prevents unnecessary string object creation
* - Named threads aid debugging with minimal overhead
*/
private static void runConcurrencyDemo() {
// BEST PRACTICE: Named thread factory for easier debugging and monitoring
// SECURITY: Thread names don't expose sensitive information
ThreadFactory namedThreadFactory = createSecureThreadFactory();

// PERFORMANCE: Fixed thread pool - threads are created once and reused
// SECURITY: Limited pool size prevents resource exhaustion attacks
ExecutorService executorPool = Executors.newFixedThreadPool(
THREAD_POOL_SIZE,
namedThreadFactory);

// CONCURRENCY: CountDownLatch provides efficient thread coordination
// Single-use barrier: count of 1 means one signal releases waiting thread
CountDownLatch completionGate = new CountDownLatch(1);

// PERFORMANCE: AtomicInteger for thread-safe counter without synchronization
// overhead
```

```
AtomicInteger upCounterState = new AtomicInteger(0);

// Create the counting tasks
Runnable countUpTask = createCountUpTask(completionGate, upCounterState);
Runnable countDownTask = createCountDownTask(completionGate);

try {
// Submit tasks to executor
// CONCURRENCY: Both tasks submitted immediately, but countDown waits on latch
logger.info("Submitting counting tasks to executor...");
executorPool.execute(countUpTask);
executorPool.execute(countDownTask);

} finally {
// BEST PRACTICE: Always shutdown executor in finally block
shutdownExecutorGracefully(executorPool);
}
}

/**
* Creates a secure thread factory with named threads.
*
* SECURITY CONSIDERATIONS:
* - Thread names don't include sensitive data
* - Uses daemon threads appropriately
* - Sets appropriate thread priority
*
* @return ThreadFactory that creates properly configured threads
*/
private static ThreadFactory createSecureThreadFactory() {
return runnable -> {
Thread thread = new Thread(runnable);

// SECURITY: Thread name provides debugging info without exposing sensitive data
thread.setName("counter-worker-" + thread.threadId());

// BEST PRACTICE: Non-daemon threads ensure tasks complete before JVM exit
thread.setDaemon(false);

// PERFORMANCE: Normal priority prevents priority inversion issues
thread.setPriority(Thread.NORM_PRIORITY);

return thread;
};
}
```

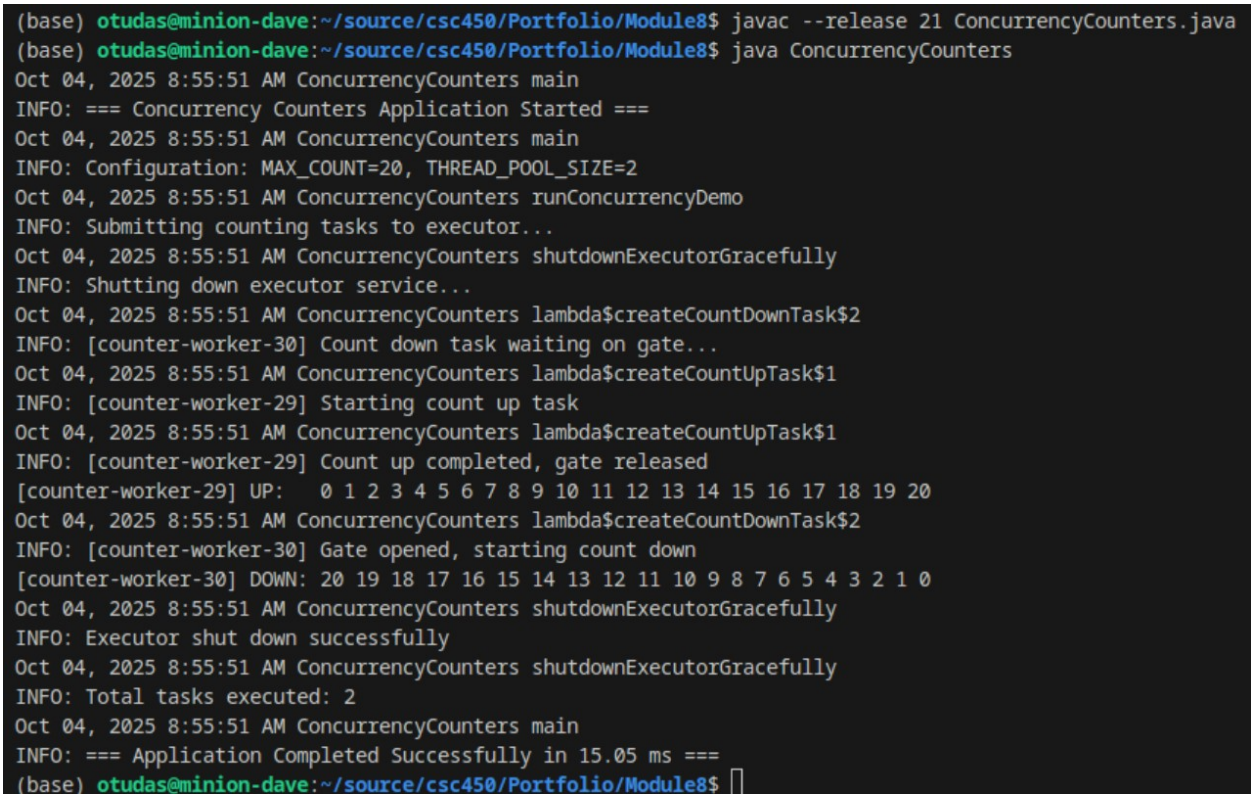
```java
/**
* Creates the count-up task.
*
* STRING VULNERABILITY ANALYSIS:
* - Uses StringBuilder (mutable) instead of String concatenation
* - Pre-allocates capacity to avoid array resizing
* - Single toString() call minimizes string object creation
*
* PERFORMANCE ANALYSIS:
* - StringBuilder: O(n) time, O(n) space - optimal for sequential appends
* - String concatenation would be O(n²) due to immutability
* - Pre-sized buffer eliminates reallocation overhead
*
* @param gate CountDownLatch to signal completion
* @param state AtomicInteger to track progress (for monitoring)
* @return Runnable task for counting up
*/
private static Runnable createCountUpTask(CountDownLatch gate, AtomicInteger state) {
return () -> {
String threadName = Thread.currentThread().getName();
logger.info("[" + threadName + "] Starting count up task");

try {
// PERFORMANCE: Pre-allocated StringBuilder prevents resizing
// SECURITY: Local scope - no shared mutable state
StringBuilder outputBuffer = new StringBuilder(STRING_BUILDER_CAPACITY);

// COUNT UP: 0 to MAX_COUNT (inclusive)
for (int i = 0; i <= MAX_COUNT; i++) {
// PERFORMANCE: StringBuilder.append is highly optimized
// Much faster than: String result = ""; result += i + " ";
outputBuffer.append(i);

if (i < MAX_COUNT) {
outputBuffer.append(' '); // Space separator
}

// MONITORING: Update progress (thread-safe atomic operation)
state.set(i);
}

// CONCURRENCY: Signal completion before output to ensure ordering
gate.countDown();
logger.info("[" + threadName + "] Count up completed, gate released");
```

```
// PERFORMANCE: Single output operation reduces I/O overhead
// STRING SECURITY: toString() creates final immutable snapshot
String output = outputBuffer.toString();
System.out.printf("[%s] UP: %s%n", threadName, output);

threadExecutionCount.incrementAndGet();

} catch (Exception e) {
logger.log(Level.SEVERE, "[" + threadName + "] Count up task failed", e);
// BEST PRACTICE: Release latch even on failure to prevent deadlock
gate.countDown();
}
};
}

/**
* Creates the count-down task.
*
* CONCURRENCY ANALYSIS:
* - Waits on CountDownLatch before starting
* - Handles InterruptedException properly
* - Preserves thread interrupt status for caller
*
* STRING VULNERABILITY MITIGATION:
* - Same StringBuilder optimization as count-up
* - No string concatenation in loops
* - Single allocation, single toString() call
*
* @param gate CountDownLatch to wait on before starting
* @return Runnable task for counting down
*/
private static Runnable createCountDownTask(CountDownLatch gate) {
return () -> {
String threadName = Thread.currentThread().getName();
logger.info("[" + threadName + "] Count down task waiting on gate...");

try {
// CONCURRENCY: Block until count-up thread signals completion
// PERFORMANCE: Efficient wait - no busy polling or sleep
gate.await();

logger.info("[" + threadName + "] Gate opened, starting count down");

// PERFORMANCE: Pre-allocated StringBuilder
```

```
StringBuilder outputBuffer = new StringBuilder(STRING_BUILDER_CAPACITY);

// COUNT DOWN: MAX_COUNT to 0 (inclusive)
for (int i = MAX_COUNT; i >= 0; i--) {
outputBuffer.append(i);

if (i > 0) {
outputBuffer.append(' ');
}
}

// PERFORMANCE: Single output operation
String output = outputBuffer.toString();
System.out.printf("[%s] DOWN: %s%n", threadName, output);

threadExecutionCount.incrementAndGet();

} catch (InterruptedException e) {
// BEST PRACTICE: Restore interrupt status
// SECURITY: Don't expose stack trace in production
Thread.currentThread().interrupt();
logger.log(Level.WARNING, "[" + threadName + "] Count down interrupted", e);

} catch (Exception e) {
logger.log(Level.SEVERE, "[" + threadName + "] Count down task failed", e);
}
};
}

/**
* Gracefully shuts down the executor service.
*
* BEST PRACTICE:
* 1. Call shutdown() to prevent new tasks
* 2. Wait for tasks to complete with timeout
* 3. Force shutdown if timeout exceeded
* 4. Handle interruption during wait
*
* SECURITY:
* - Timeout prevents resource exhaustion
* - Forces termination of runaway tasks
* - Proper cleanup prevents resource leaks
*
* @param executor ExecutorService to shut down
*/
```

```
private static void shutdownExecutorGracefully(ExecutorService executor) {
logger.info("Shutting down executor service...");

// STEP 1: Prevent new task submissions
executor.shutdown();

try {
// STEP 2: Wait for existing tasks to complete
if (!executor.awaitTermination(SHUTDOWN_TIMEOUT_SECONDS, TimeUnit.SECONDS)) {
logger.warning("Executor did not terminate within timeout, forcing shutdown");

// STEP 3: Force shutdown of running tasks
executor.shutdownNow();

// STEP 4: Wait again for forced shutdown to complete
if (!executor.awaitTermination(SHUTDOWN_TIMEOUT_SECONDS, TimeUnit.SECONDS)) {
logger.severe("Executor did not terminate after forced shutdown");
}
} else {
logger.info("Executor shut down successfully");
}

} catch (InterruptedException e) {
// BEST PRACTICE: Restore interrupt and force shutdown
Thread.currentThread().interrupt();
logger.warning("Shutdown interrupted, forcing immediate termination");
executor.shutdownNow();
}

logger.info("Total tasks executed: " + threadExecutionCount.get());
}
}
```

Execution screen shot



Comparing C++ and Java

For the purposes of writing this paper, I refactored the C++ concurrency application to be more in line with the Java application as it relates to output and logging so as to have a more accurate comparison. Overall the performance of C++ is better by a significant factor with an average runtime of 0.51ms, whereas the Java application has a runtime of 52.24ms on average. From a security perspective both have issues, but Java inherently handles variable memory better (Oracle Corporation, 2024) whereas C++ you must make sure you write it to handle variables properly or you run into buffer overflow, among other issues, that Java doesn't allow. Memory consumption overall is also handled better by C++ mainly because it does not have much overhead that comes inherently with Java's virtual machine as well as with its Garbage Collector processing. However, with that overhead you get the better inherent protections on your security that you must manually address in C++. Below I will go more details that will show that whether you should use C++ over Java is largely based on your specific needs.

Performance and memory management go hand in hand with application development, java and C++ are no different. One reason memory plays such a role in performance is due to how CPUs deal with their L1 and L2 cache. A developer has no control over when a CPU stores information into those caches but they are based on the size of the "objects" involved (Intel Corporation, 2023). Even when creating variables, like standard strings, C++ has no heap overhead (Fog, 2023), where as Java has heap overhead. This takes up more space in memory thereby playing a role in whether it stays in L1/L2 cache or not. Arguably, this is not a big impact for most applications, as you could easily argue that such a performance hit is minor. In the case of the two

applications written for this assignment, the C++ application uses approximately 80 bytes per operation, while the Java uses approximately 128 bytes and that doesn't include any overhead from garbage collection processes. This means in my implementation the Java will just as likely be kept in L1 cache as the C++, however this is a very small test with a very small amount of data. As your data grows so will your per operation footprint grow so it could have impact. While this seems significant, this is likely not the source of the significant difference in runtime however and more likely due to startup costs.

When you compile C++ you are compiling it to execute as is and immediately. The drawback is you typically have to compile it on the system, or type of system, you want it to run. For instance, I am on Linux so it gets compiled to use pthread while Windows uses WinAPI. This makes C++ a lot less portable across systems, even between Linux machines you cannot assume a C++ application will run when compiled on a different Linux system. Java on the other hand runs in a Java Virtual Machine (JVM) which has to start up every time you run a java application. This JVM includes all the functionality needed for your java app to run and is built to work on the specific platform you run on. A JVM for Linux Ubuntu is different from the Windows JVM, however, you can run your Java application on both without modifications. With C++ you cannot just compile your C++ application on a Windows machine when you wrote it on a Linux system. This JVM creates overhead for both memory usage and performance, which is the main reason we such such a huge difference between run times of our Java and C++ application. While 0.51ms is incredibly faster than 52.24ms, it is also non-noticeable to a human. So it comes down to whether performance becomes a critical factor in your

application's needs. In many cases that performance is less important than security concerns, which many enterprise applications care more about security than performance.

When building an application security is usually driven by requirements as much as overall design. Java and C++ can both do whatever it is your application needs to do to meet design specifications, but Java is clearly a safer application to use simply because you don't have to think about writing with security in mind, as much as you do with C++. With C++ you can write applications that don't handle memory corruption (eg. buffer overflows) at all and still have a fully functional application. Java, on the other hand, handles your memory management for you. While this creates overhead it also means you have don't have to spend time protecting your inputs and strings like you do in C++. This is a significant advantage when you start looking at the cost of development especially for large applications that require hundreds of hours of code development time. This is not to say Java is a panacea for developers, however. Since Java manages memory for you there is a bit of a "black box" scenario with how objects are build and maintained in memory. This can lead to vulnerabilities with denial of service type impacts, though the garbage collector is meant to help with that. C++, on the hand, is less likely as you have to be implicit with your memory and resource management. I would argue, however, that it is not enough of a reason to go with C++ as that is more an availability issue than a data leak issue and if security is a critical factor for your application Java makes it easier to meet that requirement plus there are additional security capabilities that come easier with Java.

A common attack scenario with a compromised system is privilege escalation. This means interfering with running applications and taking advantage of scenarios that

allow an attacker to assume privileges of the application running. C++ generally runs at a level in line with the context it is being run in. So when I run the app on my Linux machine is running as me. If it runs in the background I need to associate a level of security for it to run. Depending on what resources that application needs will determine the level of security it needs. In many cases C++ applications will run with elevated privileges to ensure it can do what it needs to and manage the threads it needs. Java on the other hand runs in the JVM which tends to have all of its access determined for the JVM and the application is executed in the confines of the JVM. This can create a sandbox effect where even if the Java application gets compromised the attack will only see the JVM versus having the ability to see everything on the system, unlike C++ which is not in a virtualized, or abstracted, machine.

These are only a handful of discussion points, but it is clear that choosing between Java and C++ really depends upon your requirements, budget, and frankly comfort level of your developers. Because C++ can have such a small footprint it makes it natural for when you have confines on memory or cpu power and are building for very specific situations, like hardware. Java is much more portable and has more built in security capabilities, which tends to make it quicker to develop in and easier to meet enterprise level security requirements. Personally, I have no preference towards either one. I think they both offer advantages and disadvantages making the choice driven by the requirements. Both require you to follow best practices and security guidelines if you want to maximize performance and security, you just have to choose the one that is right for your application and its requirements.

References

Fog, A. (2023). *Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms*. Copenhagen University College of Engineering. https://www.agner.org/optimize/optimizing_cpp.pdf

Intel Corporation. (2023). *Intel® 64 and IA-32 architectures optimization reference manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

ISO/IEC. (2020). *ISO/IEC 14882:2020 - Programming languages — C++*. International Organization for Standardization.

Oracle Corporation. (2024). *Java Platform, Standard Edition & Java Development Kit Version 21 API Specification*. https://docs.oracle.com/en/java/javase/21/docs/api/