Detailed Analysis CTA 8

Malyk Parker

Colorado State University Global

CSC450-1 Programming III

Reginald Haseltine

June 8, 2025

Contents

[Detailed Analysis CTA 8 3](#_Toc200281705)

[**References** 6](#_Toc200281706)

[Code Screenshot 7](#_Toc200281707)

[Output Screenshot 8](#_Toc200281708)

# Detailed Analysis CTA 8

For the Module 8 Portfolio Project result was to write two threads that count up and own. But there are some things that you must realize when doing this when it comes to the performance and security of the program. Java has a lot of built-in concurrency utilities and a garbage collector, where on the other hand C++ has no garbage collector which puts more responsibility on the programmer. In this analysis I will break down the differences between these two showing what is best and not best for each.

For java I used a fixed pool thread to avoid continuously creating and getting rid of the threads. With the use of a CoountDownLatch I could stop the down thread until the up thread was finished with its job without the lock and unlock calls that would have been needed. The garbage collectors handle the small lambdas and the running objects. On the C++ side, creating two std::thread is just as easy but without the pooling. If you wanted pooling, you would need to use something in the library or build it yourself which could become very complex. To coordinate the threads together you can use a mutex, condition\_variable, and a flag. Doing this though would require a loop for the wake ups. If we want to get technical with the performance of the program using these threads, you could look at When those threads run, both Java and C++ map to the original threads. A two-thread workload on a quad-core CPU runs smoothly, but overloading cores leads to noticeable issues. Java has an advantage here with its concurrency utilities—things like CountDownLatch hide kernel‑level park/unpark details—whereas in C++ you must manage those on your own. When it comes to memory management, Java’s garbage collector can quietly reclaim heap objects but can introduce unpredictable pauses in the code especially if you go through a lot of temporary objects. In C++ you would need to determine the destruction using RAII if you do not want any pauses to happen within the code.

Looking at the strings in both languages they all need to be paid attention to too carefully. Java has immutable string objects that are safe from unintentional changes, but if you store sensitive data in them, they can stay around in the memory until a full garbage collector sweep is done. On the other hand, in C++ std::string is mutable, so you can easily overwrite the buffer once you are done, but it is important to remember to clear it manually so that sensitive data is not lingering around. In these points we can also look at concatenating an untrusted input into queries or commands is going to be just about the same C++ and Java. I have read that most of the time with Java you can use PreparedStatements that handle the binding, while on the other hand in C++ it can often rely on manual concatenation unless you find a library that can support it.

In both examples for the project, I chose to go with primitive integer counters to get around the shared state hazards. Local variables in both threads would never work against each other meaning that there is no need for locks or atomic required. But if I wanted to a shared counter, Java does have an AtomicInteger and synchronized blocks, where on the other hand C++ would depend on the std::atomic or explicit mutexes. Another thing to keep in mind is looking at the permissions perspective of things. I learned that Java once had a SecurityManager to limit the thread creation at a time, but it is no longer supported in my environment and that now it more load on the OS level to control this along with the containers. In C++ the applications have whatever privileges are given. So, this means that the security boundary is mainly on the programmer for the configuration and other tools that can be used.

Another angle to look at this is with exception handling and thread interruption. With Java catching InterruptedException and restoring the threads silent failures ensure that high level workflows can detect and react to the cancellations that could happen. Logging the frameworks should be configured to mask or get rid of sensitive data avoiding the risk of secrets appearing in log files. With C++ exception handling across threads requires careful thought on uncaught exceptions in which a worker thread can terminate the entire program unless caught within std::thread. When using condition variables, false wakeups or missed notifications must be handled to avoid deadlocks or unintended busy loops, both of which can expose performance issues.

Looking at a complete comparison between the two, both languages have similar costs and interactions with the kernel, but they have very different approaches to memory, security, and synchronization. I think that Java would win on the simplicity of having built in tools for safety, bounds checking, concurrency constructs, and managed memory to reduce the risk of buffer overflows, race conditions, and easier to debug. When it comes to C++ it can give you maximum control, meaning there is no garbage collector, you have direct access to the OS primitives, but with all the control you are given that can come at a cost. There is a lot more responsibility and pitfalls that you can run into when using C++.

Overall, for day-to-day development where safety and productivity are of the upmost importance, along with looking at a smaller learning curve, and stronger resources Java takes the crown with that idea. When it comes to high performance and a lot of resources are needed, C++ is the language that outperforms Java. But with C++ you must ensure that you are well versed with the memory and synchronization that comes with it.

# **References**

[Secure Coding Guidelines for Java SELinks to an external site.](https://www.oracle.com/java/technologies/javase/seccodeguide.html)

Goffi, A., Gorla, A., Mattavelli, A., & Pezzè, M. (2023). [Intrinsic redundancy for reliability and beyondLinks to an external site.](http://software.imdea.org/~alessandra.gorla/papers/Goffi-Redundancy-PAUSE17.pdf). iMdea. <https://software.imdea.org/~alessandra.gorla/papers/Goffi-Redundancy-PAUSE17.pdf>

Chapters 7 & 8 in [Introduction to programming using JavaLinks to an external site.](http://math.hws.edu/javanotes/)

Gassner, D. (2019, November 19). [Handle exceptions with try/catch from java 11+ essential trainingLinks to an external site.](https://www.linkedin.com/learning-login/share?account=2245842&forceAccount=true&redirect=https%3A%2F%2Fwww%2Elinkedin%2Ecom%2Flearning%2Fjava-11-plus-essential-training%2Fhandle-exceptions-with-try-catch&shareId=7dc86f6b-da2a-4934-bdd0-e66e83f0557d). [Video]. LinkedIn Learning.

# 

# Git Repository Link

<https://github.com/Malypar/Mod8-Portfolio-Project.git>

# Code Screenshot

A screen shot of a computer program

AI-generated content may be incorrect.

# Output Screenshot

A screen shot of a computer

AI-generated content may be incorrect.