**Portfolio**

Joshua Williams

Colorado State University Global

CSC450: Computer Programming III

Dr. Jack Li

August 3rd 2025

**Portfolio**

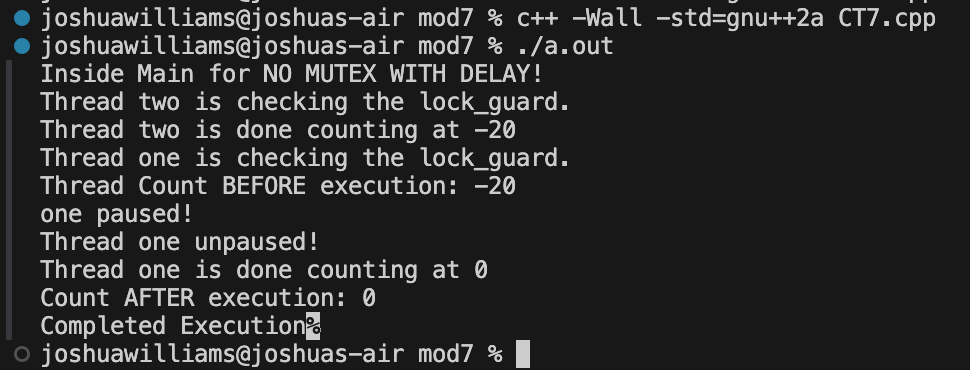
This project covers multi-threading, as well as secure programming in C++ and Java.

**Working with Threads in C++**

Last week we were tasked with writing up a demonstration of concurrency in C++. Concurrency can be achieved through the use of threads. Threads are processes or execute blocks that can be run independently from one another. Threads can also be run at the same time, also known as multi-threading, which allows for concurrent execution (Geeks for Geeks, 2025). Below is my output code for running two threads. The first or one thread is supposed to increase our count variable by 20, and I gave it a 3 second delay to force it to be slower. The second or two thread is meant to decrease the count variable by 20. The second thread has no delay. Using threads opens up our application to new vulnerabilities such as race conditions. This happens when multiple threads are running at once and they try to access the same resource, and write to that resource at the same time. This can lead to unexpected results. Mutexes can be used to prevent this. Below in Figure 1 is my output of the two threads running at once.

**Figure 1**

*Screenshot of test code without Mutex or Lock Guard.*

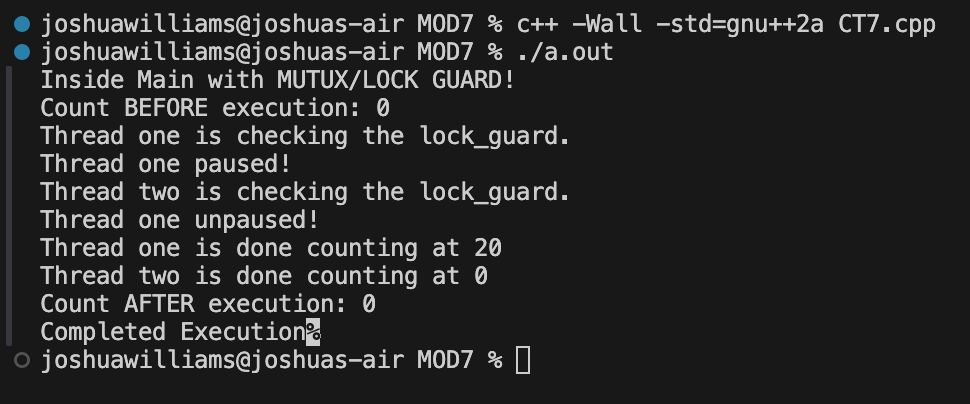


*Note*. I commented out the mutex lines in my function.

As you can see the second thread finishes, before the first and displays a -20, and then the first thread runs and displays the 0. In Figure 2 I give my code a Mutex to use with lock guard.

**Figure 2**

*Screenshot of test code with Mutex and Lock Guard.*



*Note*. Code with Lock guard forces the second thread to wait.

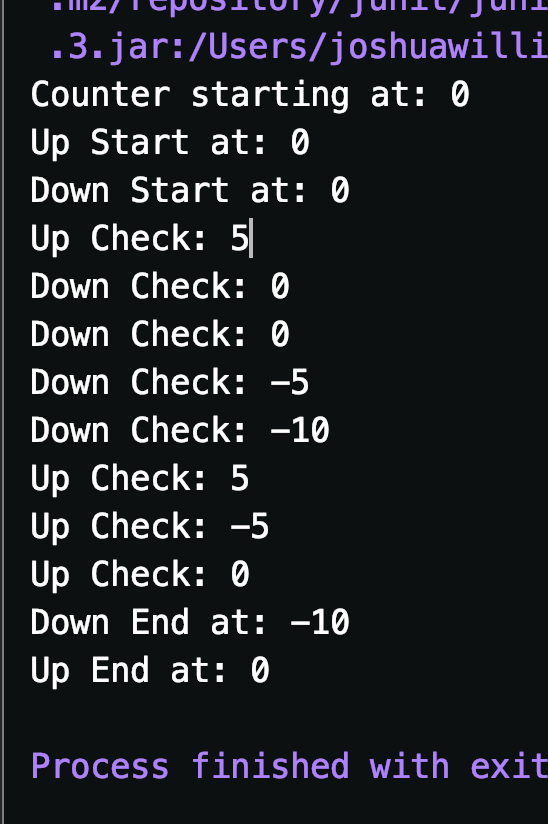
Lock guard forces threads that hit that point to check if any other threads have locked down that section of code. If the lock is enabled, then that means there is another thread currently using it, and it will have to pause and wait until the mutex lock is unlocked again for the next thread. There are other options to lock down critical sections of code, such as just using mutexVar.lock() and .unlock() to manually lock/unlock code blocks. However lock guard simplifies this process, by automatically locking and unlocking for me, as needed.

**Working with Threads in Java vs C++**

I also created some test code to demonstrate concurrency in Java. Below in Figure 3 is my output code, when running two threads concurrently without a mutex.

**Figure 3**

*Screenshot of test code without mutex.*

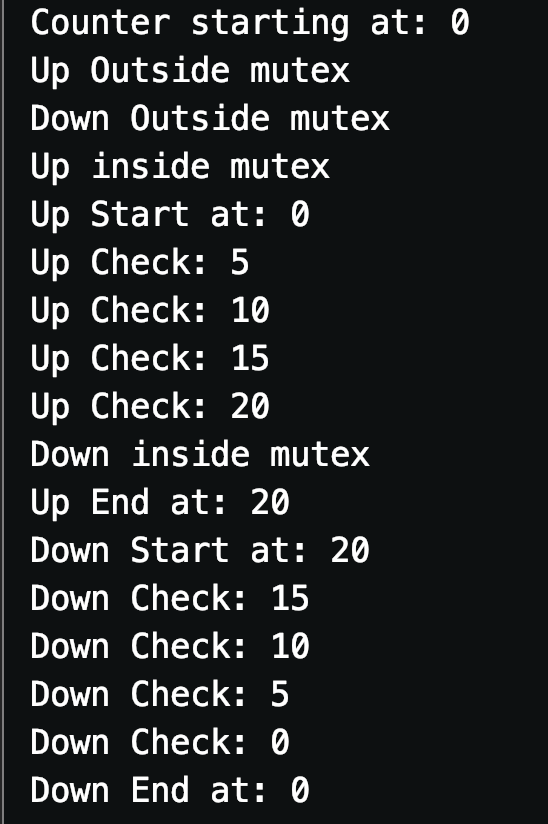


*Note*. Both threads were accessing the count object at the same time.

Running both threads without a mutex could lead to unexpected behavior where one of the thread’s actions may not be recorded, and we could end on a number other than 0. If I had used a while loop, rather than a for loop, both threads would continue to run, until one thread’s while condition could be broken. In Figure 4, I show my output code with a mutex, using the synchronized keyword.

**Figure 4**

*Screenshot of test code with Mutex using synchronized keyword .*



*Note*. I added log messages to show where the mutex check begins.

Similar to C++, there are multiple ways to implement a mutex. Both languages offer the same functionality, even including semaphores, which allow multiple threads to access critical sections at the same time.

**C++ vs Java Performance**

Both the C++ and Java programs ran very quickly. However after adding timers to both programs, the C++ program was completing execution within less than a millisecond, averaging 0.2 to 0.5 milliseconds. My Java program on the other hand was coming in at 5 to 9 milliseconds. Both languages are compiled languages, but the process from human readable code to executable code is different between them. Java has to compile the code into bytecode, and send that code to the JVM, which then processes that code into machine code for execution. C++ on the other hand, does not include a middle man, and complies into machine code (Geeks for Geeks, 2025).

**Vulnerabilities associated with Strings in C++ and Java**

C++ does not have a true primitive “string” object, like most other programming languages. Strings are actually character arrays, and can be accessed in the exact same way. This leaves strings open to all kinds of issues, such as buffer overflows, out of bounds errors, and more (Kazankova, n.d.). C++ does offer a “string” object through its standard library (std). When using a string created from the stand library, you can avoid most of these vulnerabilities, since the string object will resize itself as needed when adjusting the character array. Java also has all kinds of string vulnerabilities, with a large amount coming from injection attacks (Codacy, 2025). Whether they are sql injection or command injection attack, the mitigation strategy is the same and involves input validation and sanitization (Codacy, 2025). You can also limit your attack paths by following least privilege. For example where I work, each service and application has their own database account, so one service cannot make changes to another’s tables if the account or service were to be compromised.

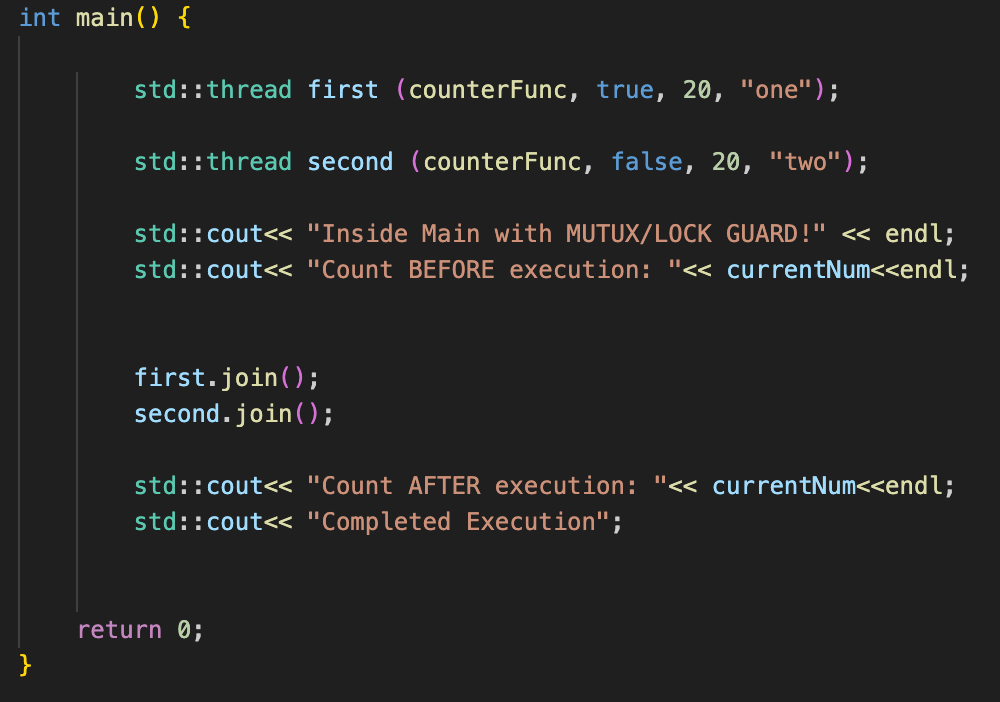
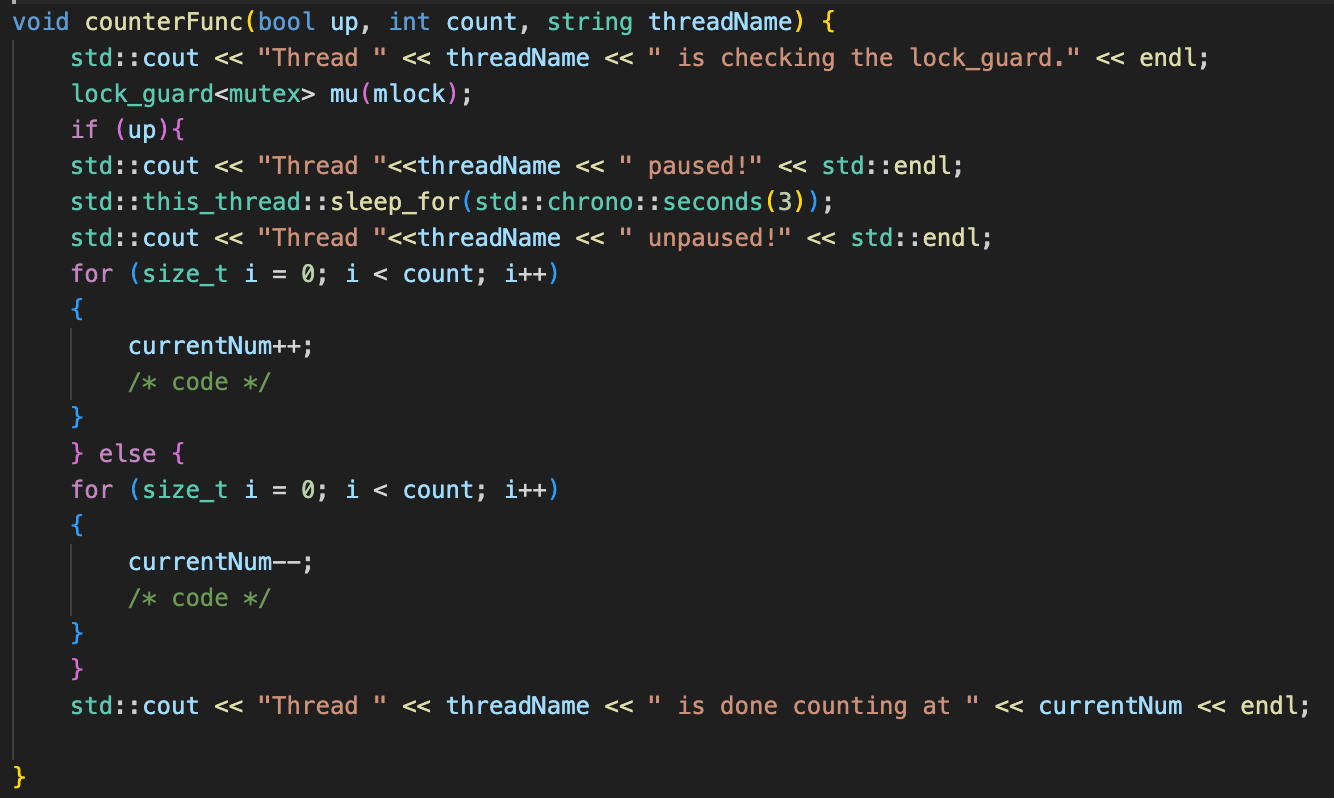
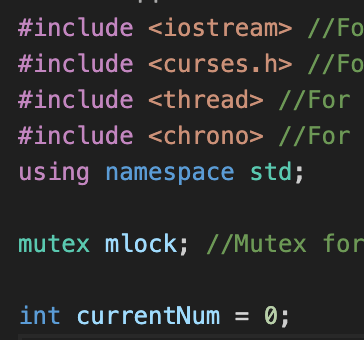
**Security of the other data types shown in C++**

The other data types in C++ and Java also have many vulnerabilities. Two of the most prevalent ones being out of bounds read and write. An out of bounds write happens when a program attempts to write out of the memory space allocated for the variable it was trying to write to. This can lead to crashing or even malicious code execution (Kazankova, n.d.). Out of bounds read vulnerabilities can lead to data exposure and crashes. To help mitigate both of these vulnerabilities, the programmer should handle exceptions in a safe manner, without exposing the inner workings of the program. Both C++ and Java offer custom exceptions, as well as exception handling.

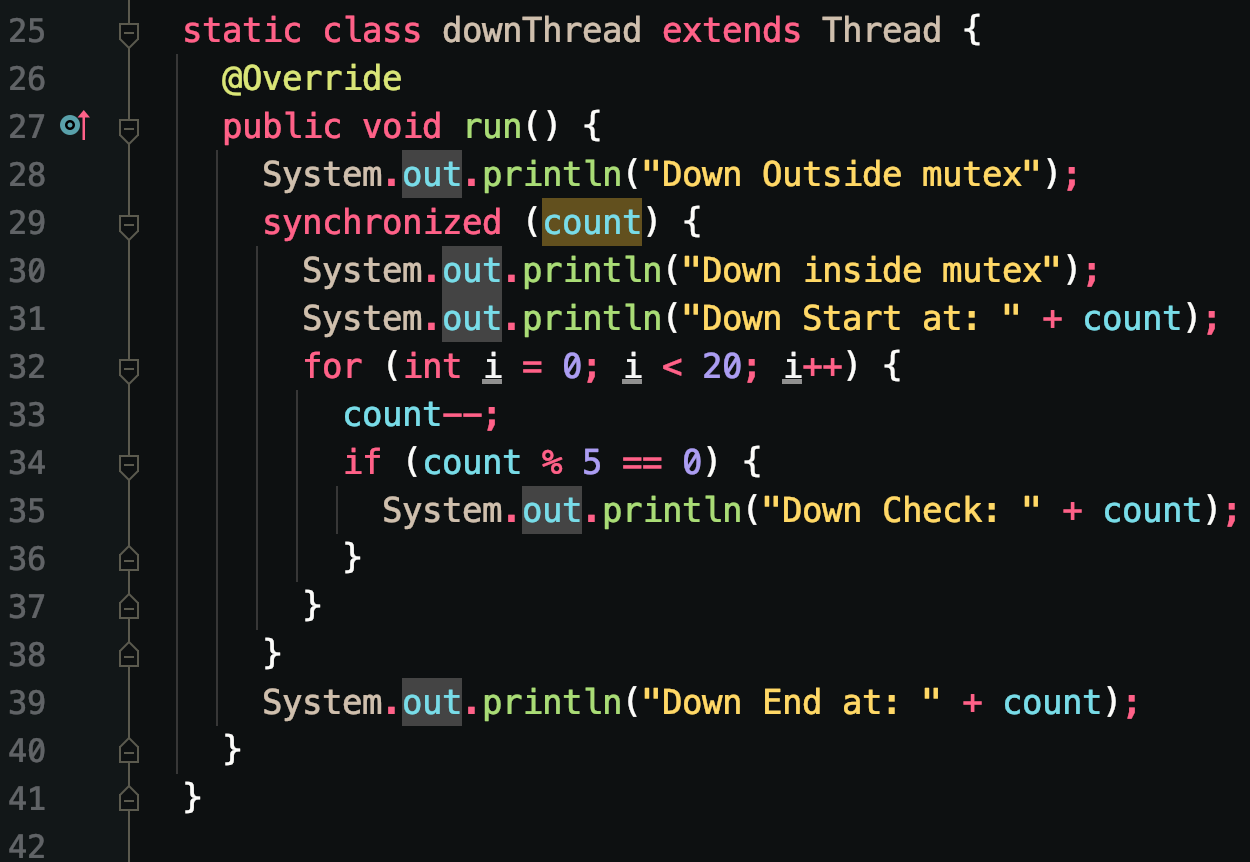
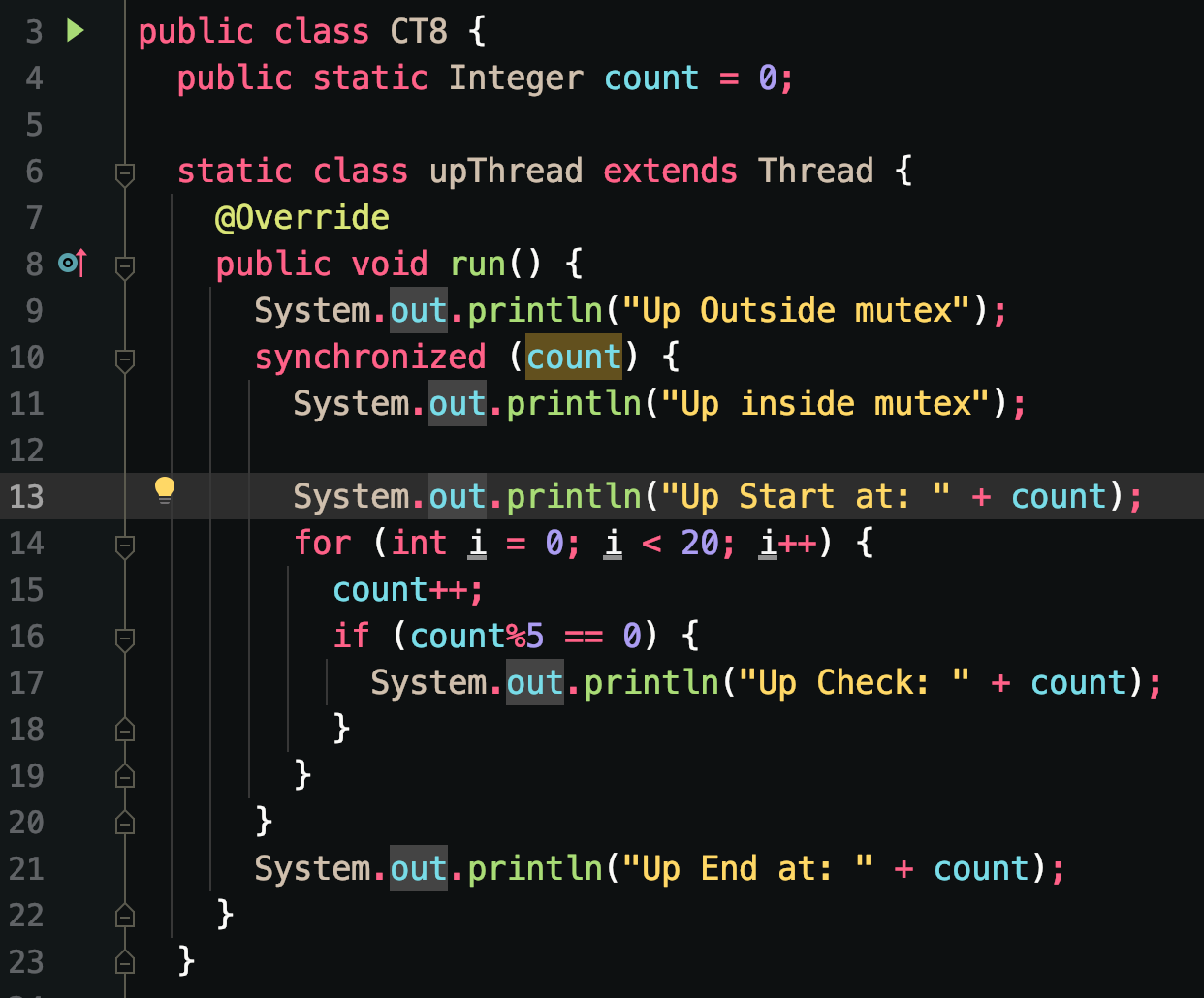
**Conclusion**

C++ and Java both share vulnerabilities as do other major programming languages. These vulnerabilities can be mitigated through a variety of techniques, as well as attention to detail. The smallest mistakes have led to some of the largest data breaches. It is important to follow the most up to date coding standards to ensure that your application is not exposed to preventable threats. If a threat is not preventable, then it is important to at least mitigate the actions of that threat, and lower the impact of an attack.

**Code Review C++**



**Code Review Java**



**Github Links**

C++ code: <https://github.com/Joshua-Williams8/Java-SE-8-notes/tree/main/src/CSC450/MOD7/MOD7CT>

Java code: <https://github.com/Joshua-Williams8/Java-SE-8-notes/blob/main/src/CSC450/MOD8/Portofolio/CT8.java>

**References**

Codacy. (2025, May 7). *11 Common Java Vulnerabilities and How to Avoid Them*. codacy.com. <https://blog.codacy.com/java-vulnerabilities>

Kazankova, N. (n.d.). Top Six Most Dangerous Vulnerabilities in C and C++. code-intelligence.com. <https://www.code-intelligence.com/blog/most-dangerous-vulnerabilities-cwes-in-c-2025>

Geeks for Geeks. (2025, May 15). *Multithreading in C++*. geeksforgeeks.org. <http://geeksforgeeks.org/cpp/multithreading-in-cpp/>

Geeks for Geeks. (2025, June 23). *Why Java Language is Slower Than CPP for Competitive Programming?*. geeksforgeeks.org. <https://www.geeksforgeeks.org/java/why-java-language-is-slower-than-cpp-for-competitive-programming/>

Tutorials Point. (n.d.). *C++ Concurrency*. tutorialspoint.com. <https://www.tutorialspoint.com/cplusplus/cpp_concurrency.htm>