**Final Project: Concurrency Concepts Java & C++ Comparison**

Jordon Paynter

Department of Computer Science, Colorado State University Global

CSC 450: Computer Programming II

Professor Farhad Bari

November 5, 2023

Contents

[**Course Recap** 3](#_Toc150078630)

[**Introduction** 3](#_Toc150078631)

[**Module 1: Create a C++ Application** 3](#_Toc150078632)

[**Module 2: String Handling in C++** 3](#_Toc150078633)

[**Module 3: Function and Pointer Exploits** 3](#_Toc150078634)

[**Module 4: Integer Security in C++** 4](#_Toc150078635)

[**Module 5: Concurrency and File I/O in C++** 4](#_Toc150078636)

[**Module 6: Java Security** 4](#_Toc150078637)

[**Module 7: Advanced Java Concepts** 4](#_Toc150078638)

[**Conclusion of Course Recap** 5](#_Toc150078639)

[**Java Concurrency Program** 6](#_Toc150078640)

[Execution of the Java Code counting. 6](#_Toc150078641)

[**Java code program analysis** 7](#_Toc150078642)

[**Introduction** 7](#_Toc150078643)

[**Performance Issues with Concurrency** 7](#_Toc150078644)

[**Vulnerabilities Exhibited with the Use of Strings** 7](#_Toc150078645)

[**Security of the Data Types Exhibited** 8](#_Toc150078646)

[**Conclusion** 8](#_Toc150078647)

[**Java Source Code** 9](#_Toc150078648)

[Source Code: ConcurrencyCounting.java 9](#_Toc150078649)

[Source Code Continued: ConcurrencyCounting.java 10](#_Toc150078650)

[**Comparison of Java and C++ Concurrency Implementations** 11](#_Toc150078651)

[**Introduction** 11](#_Toc150078652)

[**Thread Synchronization and Contention** 11](#_Toc150078653)

[**Sleep Intervals** 12](#_Toc150078654)

[**Thread Sequencing** 12](#_Toc150078655)

[**Security Aspects** 13](#_Toc150078656)

[**Use of Strings** 13](#_Toc150078657)

[**Security of Data Types** 14](#_Toc150078658)

[**Conclusion** 14](#_Toc150078659)

[References 15](#_Toc150078660)

# **Course Recap**

## **Introduction**

Welcome to this quick review of the weekly modules we’ve covered throughout this course. We’ll revisit each module’s key concepts and topics, briefly overviewing what we’ve learned. Let’s dive in!

## **Module 1: Create a C++ Application**

We learned about the essential concepts in C++ programming. We covered data types, including int, float, double, char, and bool, which are crucial for defining variables. We also discussed control structures like if statements, for, while, and do-while loops, and switch statements to control program flow. Additionally, we touched upon software threats such as buffer overflows and injection attacks and the importance of security concepts like input validation, secure coding practices, and vulnerability protection.

## **Module 2: String Handling in C++**

We explored the string data type and how to declare, initialize, and manipulate string objects. We discussed the potential for buffer overflow vulnerabilities in C++ applications, mainly related to string handling. We also examined runtime solutions for mitigating buffer overflows, emphasizing bounds checking and using safer string functions. This module highlighted the significance of proper string handling techniques to enhance security.

**Module 3: Function and Pointer Exploits**

We delved into function and pointer exploits in C++ applications. We identified potential vulnerabilities associated with function pointers and their exploitation by attackers. We also learned how to override function pointers to ensure security and discussed exception handlers, including try-catch and throw, for error handling.

## **Module 4: Integer Security in C++**

We identified factors affecting integer security, such as overflow, implicit type conversion, and signed vs. unsigned integers. We discussed exception errors in integer operations and common integer vulnerabilities, including buffer overflows and injection attacks. Formatting vulnerabilities related to the misuse of formatting functions like printf were also highlighted, with suggestions for mitigation.

## **Module 5: Concurrency and File I/O in C++**

We explored concurrency vulnerabilities, including race conditions, deadlocks, thread synchronization, and resource contention. File input/output operations were discussed, emphasizing reading from and writing to files, stream operations, and error handling. We also examined file access control mechanisms and potential file directory vulnerabilities in C++ applications.

## **Module 6: Java Security**

We shifted our focus to Java security. We discussed techniques for securing sensitive data, including encryption, secure storage, access controls, and secure communication. Access permissions in Java, achieved through access modifiers like public, private, and protected, were explained. We also learned about minimizing privilege code access, addressing noncompliant code, and developing solutions for application compliance to enhance Java application security.

## **Module 7: Advanced Java Concepts**

We learned about variable scope minimization’s benefits, access mechanisms for classes using access modifiers, and techniques for improving concurrency in Java applications. Cyclic dependencies between Java packages and the implementation of user-defined exceptions were also explored.

## **Conclusion of Course Recap**

This course has provided us with a comprehensive understanding of programming concepts and security practices in both C++ and Java. From Module 1’s introduction to fundamental data types and control structures to Module 7’s exploration of advanced Java concepts, we have covered various topics for building secure and efficient applications. We have learned to identify and mitigate various security threats, vulnerabilities, and best practices in both languages, ensuring we are better equipped to develop robust and secure software. As we conclude this course, we have gained valuable insights that will undoubtedly prove invaluable in our future endeavors as programmers and developers.

# **Java Concurrency Program**

**Figure 1.**

## Execution of the Java Code counting.

A screenshot of a computer program

Description automatically generated

## **Java code program analysis**

### **Introduction**

This Java application showcases fundamental concurrency concepts by creating two threads – one for counting up and another for counting down. This analysis considers performance issues, potential vulnerabilities related to string usage, and the security of data types.

### **Performance Issues with Concurrency**

The code employs synchronized methods (**incrementCounter** and **decrementCounter**) and synchronized blocks with a common mutex (**ConcurrencyCounting.class**) to protect the shared counter variable, preventing race conditions. However, this synchronization strategy can lead to performance bottlenecks. Thread two waits on thread one to finish counting, causing resource wastage as it needs to wait. This scenario allows one thread to count up and down while the other handles another resource.

**Thread.sleep(100)** introduces artificial delays, which can impact performance. These delays are primarily for improving output readability but might not be advisable in a real-world application. They consume resources unnecessarily.

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

This code primarily utilizes strings for debugging and informational purposes without involving string manipulation or user input. As such, it doesn’t exhibit vulnerabilities related to strings. However, if there were string input from users, sanitizing it for security purposes would be crucial.

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

Using an integer data type for the counter is appropriate for this scenario. The program incorporates a **MAX\_COUNT** constant to ensure the counter remains within the upper bound. Mutexes are correctly employed for secure synchronization, ensuring data integrity during concurrent access.

### **Conclusion**

This Java application effectively demonstrates fundamental concurrency concepts by creating two threads that count up and down concurrently. It addresses performance considerations, security of data types, and synchronization using mutexes.

## **Java Source Code**

**Figure 2a.**

### Source Code: ConcurrencyCounting.java

A screenshot of a computer program

Description automatically generated

**Figure 2b.**

### Source Code Continued: ConcurrencyCounting.java

A screen shot of a computer code

Description automatically generated

# **Comparison of Java and C++ Concurrency Implementations**

## **Introduction**

We will compare the Java and C++ implementations of a concurrent counting application. We’ll evaluate their performance, concurrency handling, and security aspects. Both implementations aim to count up to a specified maximum value and then count down to zero using two threads while considering potential performance bottlenecks and security concerns.

## **Thread Synchronization and Contention**

**Java ConcurrencyCounting.java:** Synchronized methods **incrementCounter** and **decrementCounter** are used to ensure that only one thread can access the shared **counter** variable at a time. This approach prevents data races and ensures data integrity. However, it also leads to performance bottlenecks as both threads synchronize on the same object, **ConcurrencyCounting.class**. Both threads cannot run concurrently and execute sequentially, limiting the benefits of multithreading.

**C++ ConcurrencyCounting.cpp:** The C++ version employs a mutex to protect the shared **counter** variable. The **lock\_guard** automatically releases the mutex when the function scope ends, ensuring mutual exclusion. Like the Java version, the C++ implementation also faces thread contention issues because it waits for the first thread to finish counting up before the second thread can start counting down. This behavior restricts concurrency and may lead to performance inefficiencies.

In C++, ensuring thread safety is more intricate compared to Java. Achieving the same level of thread safety in the C++ implementation involves creating and meticulously managing mutex objects. It’s essential to emphasize that the provided example is relatively straightforward, and real-world scenarios often entail considerably more intricate thread safety requirements. (katlı, 2023)

### **Sleep Intervals**

**Java:** The Java version introduces artificial delays with **Thread.sleep(100)** to slow down the threads for improved output readability. While this delay does not directly impact the application’s correctness, it unnecessarily consumes system resources.

**C++:** The C++ version also uses sleep intervals with **this\_thread::sleep\_for(chrono::milliseconds(100))** to simulate work and allow concurrent execution. However, it introduces delays less frequently than the Java version, resulting in more efficient resource utilization.

### **Thread Sequencing**

**Java:** In the Java version, there is an attempt to handle thread sequencing using a **while** loop and checking the **counter** value. This approach ensures that one thread counts up. In comparison, the other counts down but may not be foolproof and could lead to a condition where one thread keeps waiting indefinitely, causing resource locking and affecting performance.

Initially, Java primarily relied on monitors utilizing the synchronized keyword and the wait-and-notify mechanism for managing thread synchronization. However, this approach posed significant challenges regarding signaling between threads, often resulting in “missed signals.” Recognizing these shortcomings, Java’s designers introduced a comprehensive concurrency package that addressed these concerns and included features like non-blocking data structures. (Perera, 2018)

**C++:** The C++ version employs a different approach to handle thread sequencing, using a **troubleTracker** to detect if the other thread is taking too long to count up. However, this mechanism does not prevent a deadlock if one thread fails to proceed as expected.

A simple search yields C++ 22 concurrency libraries and 6 C concurrency libraries, showcasing the abundance of powerful tools available. These libraries offer a wide range of cutting-edge technologies. However, their diversity in APIs can be a challenge, leading to a scarcity of highly proficient programmers with any given API. Furthermore, C++ demands a higher level of developer involvement for fine-tuning, capitalizing on the flexibility and control it provides. (Perera, 2018)

## **Security Aspects**

### **Use of Strings**

In both Java and C++ implementations, strings are primarily used for debugging and informational purposes, such as printing messages to the console. It’s important to note that neither implementation involves user input or string manipulation. Therefore, neither version is vulnerable to issues related to strings.

These strings are used to provide informative output for debugging and monitoring the application’s progress. Since there is no interaction with user-generated strings or data, the risk of vulnerabilities related to strings, such as input validation or injection attacks, is mitigated.

Mr. Falconer did point out that “Historically, logs are often the target of data breaches or the source of accidental data leaks. Keeping sensitive data out of your logs is a simple way to address this issue. Attacks are going to happen, but by keeping sensitive data out of your logs, you’re significantly reducing the value of any data that gets compromised.” (Falconer, 2022)

Both implementations ensure that the strings used for debugging and informational purposes do not inadvertently expose sensitive information. This practice aligns with security best practices by limiting the exposure of potentially sensitive data in program output.

### **Security of Data Types**

The Java version uses an **int** data type for the **counter**, which is appropriate for this scenario. The program employs synchronized methods to ensure data integrity during concurrent access. Additionally, it uses a constant **MAX\_COUNT** to set an upper bound for the counter, reducing the risk of integer-related vulnerabilities.

The C++ version also uses an **int** data type for the **counter**, and it is protected using a mutex to ensure data integrity. Like the Java version, it includes a constant **MAX\_COUNT** to set an upper bound for the counter, minimizing the risk of integer-related vulnerabilities.

## **Conclusion**

Both the Java and C++ implementations showcase fundamental concurrency concepts but face similar issues related to thread contention, limiting the potential parallelism of multithreading. While both versions are equally secure in terms of data types and string usage, they may benefit from alternative concurrency strategies to achieve better parallelism and reduce contention, particularly in scenarios where performance is a critical factor.

# References

Falconer, S. (2022, May 4). *Keep sensitive data out of your logs: 9 best practices*. Keep Sensitive Data Out of Your Logs: 9 Best Practices. https://www.skyflow.com/post/how-to-keep-sensitive-data-out-of-your-logs-nine-best-practices

katlı, oğuzhan. (2023, September 25). *Unlocking the power of Java synchronized blocks in C++*. Medium. https://oguzhankatli.medium.com/unlocking-the-power-of-java-synchronized-blocks-in-c-8fe6fbbc0d66#:~:text=However%2C%20C%2B%2B%20doesn’t%20have,to%20write%20thread%2Dsafe%20code.

Perera, S. (2018, July 3). *Draconian, free, or nanny state: Concurrency ideologies in Java, C#, C, C++, go, and rust*. Draconian, free, or nanny state: Concurrency ideologies in Java, C#, C, C++, Go, and Rust. https://www.freecodecamp.org/news/concurrency-ideologies-of-programming-languages-java-c-c-c-go-and-rust-bd4671d943f/