**Analysis of Concurrency in Java**

In this paper, I provide a detailed analysis of the program I created to demonstrate concurrency in Java. The program uses one thread to increment a counter to twenty, then subsequently uses a second thread to decrement the counter to zero. I provide an overview of the performance issues related to concurrency, followed by a discussion of string vulnerabilities and how they are avoided in the program. In the final section, I address the security of data types used in the program.

**Concurrency Performance Concerns**

Concurrency in Java can introduce performance concerns such as data races, deadlocks, and thread contention. The program I designed to increment and decrement a counter avoids and mitigates concurrency performance issues by using synchronized methods as well as the notify and wait methods.

**Data Race Conditions**

One potential performance issue is a data race. A data race occurs when multiple threads access shared data and attempt to modify it simultaneously, potentially causing undefined behaviors or runtime exceptions (Burcea, 2024). These undefined behaviors could lead to security vulnerabilities by allowing attackers to manipulate data in unexpected ways. My program avoids race conditions by using the synchronized key word in SharedCounter’s methods. This synchronization ensures the increment and decrement methods cannot run concurrently for a single instance of SharedCounter. Therefore, only one thread can access the shared data at a time.

**Deadlock**

Another potential performance issue is deadlock. Deadlock occurs when two or more threads block each other, with each waiting for a resource held by another thread (Burcea, 2024). Deadlock introduces a potential security concern, as it could be used to create a denial of service. Because there is only one shared resource in my program, there is no possibility for a deadlock to occur.

**Thread Contention**

Thread contention occurs when threads are waiting for access to a synchronized block of code. This creates a performance bottleneck. In my program, this occurs while the decrementThread waits for incrementThread to finish executing. This is an intended feature of my program. However, in a more complex application thread contention can cause degraded performance due to bottlenecks such as this.

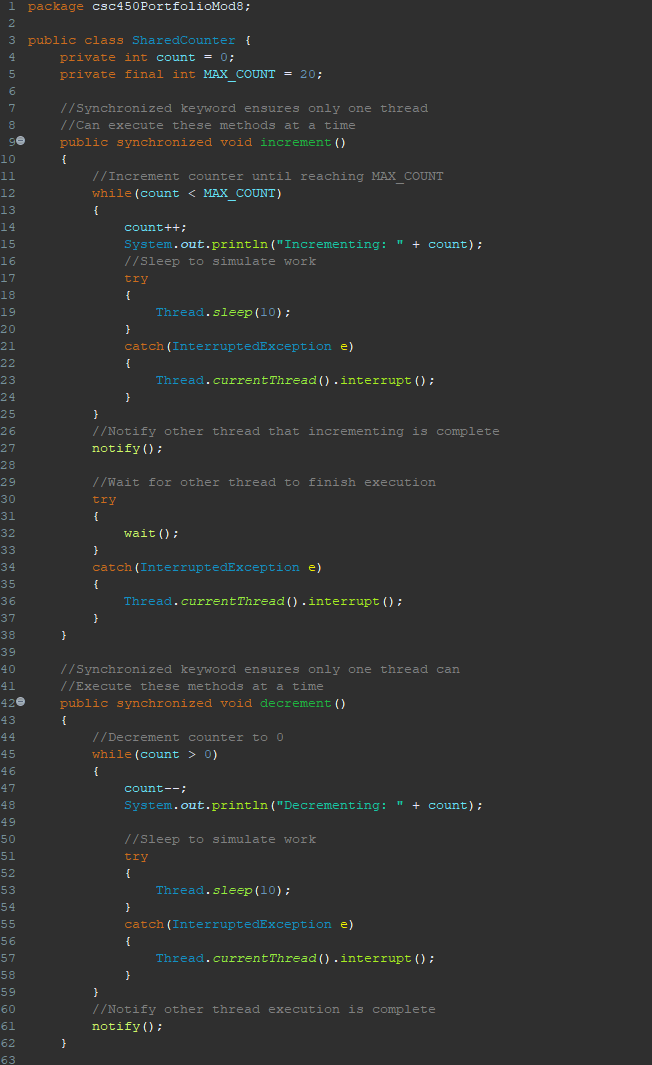
**String Vulnerabilities**

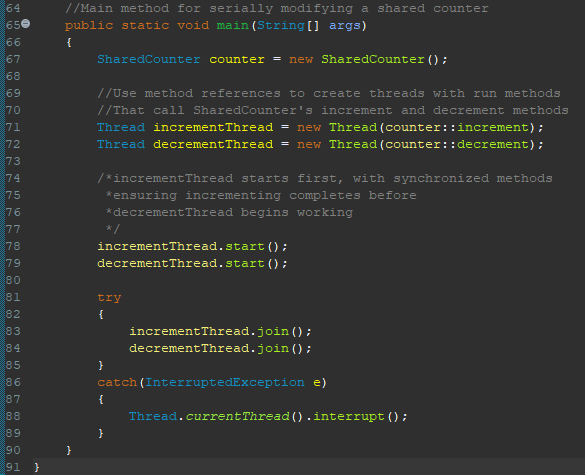
Just as in C++, Java programs can have a format string vulnerability. In Java, this vulnerability exists when unsanitized user input is passed into the ‘System.out.printf(..)’ function with a conversion specifier (OWASP, 2020). This would result in the program crashing with an IllegalFormatException. Because the ‘printf’ method is not used and my program does not accept user input, it is safe from string vulnerabilities.

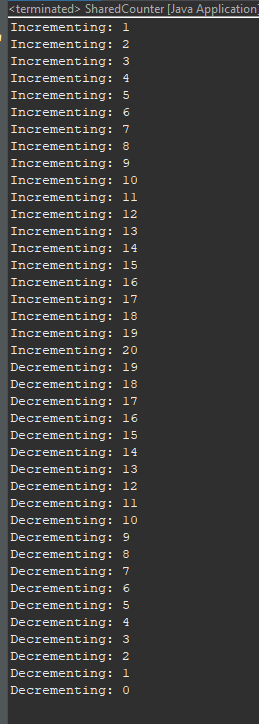
**Data Type Security**

The data types utilized in my program include the two integers in the ‘SharedCounter’ class. These integers could be vulnerable to integer overflow. In my program, the integers never exceed a value of 20, so this is not a concern. The primitive data type also does not provide thread-safe operations. The synchronized methods used to modify the counter in my program mitigate this. However, an alternative option that could enhance performance would be using the AtomicInteger class. This class provides a way to ensure the counter is thread-safe without synchronization. Due to the small performance requirements of my program, I decided to use synchronized methods instead.

**Source Code**







**References**

Burcea, C. (2024, January 8). *Common concurrency pitfalls in Java*. Baeldung. <https://www.baeldung.com/java-common-concurrency-pitfalls>

Oracle. (n.d.). *Deadlock*. The JavaTM Tutorials. <https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html>

OWASP. (2020). *Testing for Format String Injection*. WSTG - Stable | OWASP Foundation. <https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/07-Input_Validation_Testing/13-Testing_for_Format_String_Injection>