**Module 8: Portfolio Project**

Brandy Martinez

Colorado State University Global

CSC450: Programming III

Jack Li

04/06/2025

**Concurrency Concepts**

START Program

DEFINE class CountUp:

METHOD run:

FOR i FROM 1 TO 20:

DISPLAY "Count Up: " + i

WAIT for 100 milliseconds

END FOR

DEFINE class CountDown:

METHOD run:

FOR i FROM 20 TO 0:

DISPLAY "Count Down: " + i

WAIT for 100 milliseconds

END FOR

MAIN METHOD:

CREATE instance of CountUp

CREATE instance of CountDown

CREATE thread1 to run CountUp

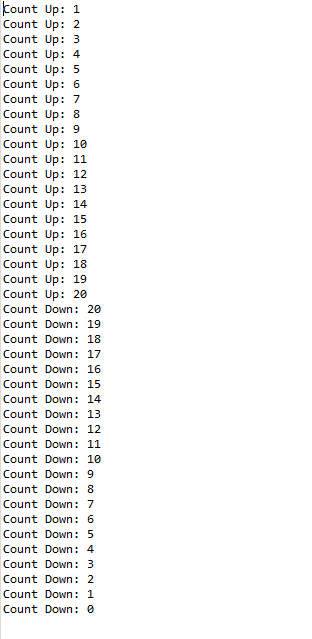
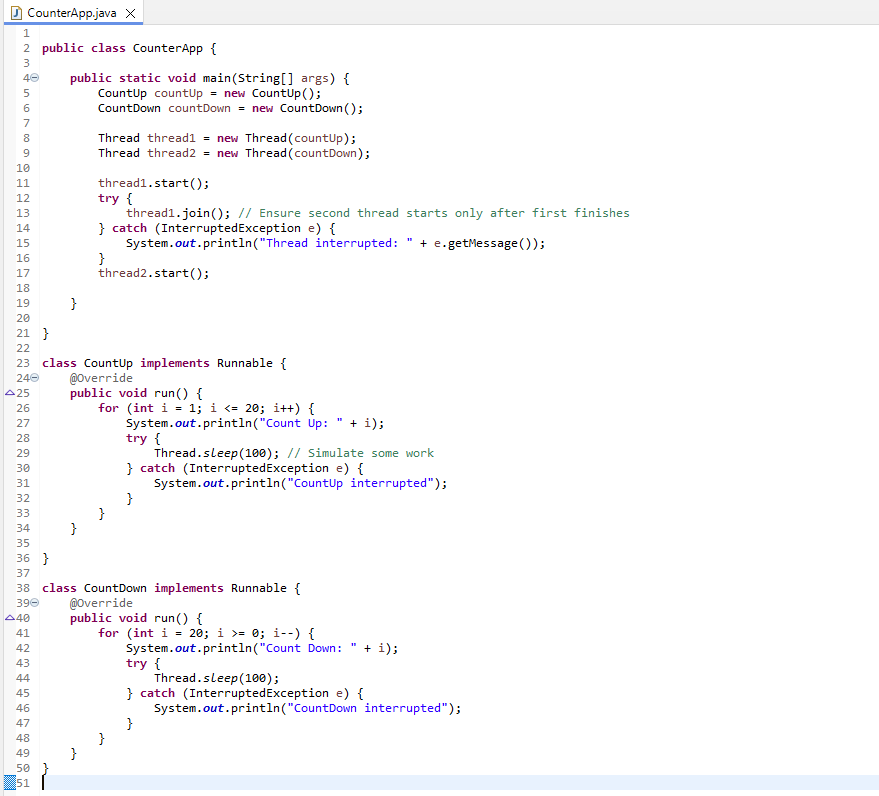
CREATE thread2 to run CountDown

START thread1

WAIT until thread1 finishes (join)

START thread2

END Program



The CounterApp Java program demonstrates essential concurrency concepts using multithreading. By utilizing two threads to count numbers sequentially, one counting up and the other counting down, it illustrates how Java handles parallel execution and thread management. Several important topics emerge from this implementation, including performance considerations, exception handling, and data security concerns especially in relation to thread behavior and string manipulation.

The core concept in this program revolves around concurrency, implemented through the Runnable interface and Java’s Thread class. Two classes, CountUp and CountDown, implement Runnable, each defining a run() method that prints numbers in a loop with slight delays (Thread.sleep(100)), simulating work. In the main() method, two threads are created: one for counting up and one for counting down. However, only one thread is started at a time. The join() method is used to ensure the first thread (CountUp) completes before the second (CountDown) begins. This highlights the importance of thread coordination especially when the order of operations is critical. Without join(), both threads would run simultaneously potentially leading to mixed output which may be undesirable depending on the case.

Concurrency, while powerful, must be handled carefully to avoid performance bottlenecks or erratic behavior. In this example, because the threads are run sequentially rather than in true parallel, it doesn’t fully utilize the benefits of concurrent processing. In real-world applications, improper synchronization or overuse of threads can result in context switching overhead, memory leaks, or even thread starvation. In larger applications, developers must also be cautious of race conditions, where multiple threads access and modify shared resources without proper synchronization. While this specific example avoids shared data and is thread-safe by design, it serves as a foundation for understanding the broader implications of multithreaded applications.

Robust exception handling is another concept demonstrated in this application. Each thread uses a try-catch block around Thread.sleep() to handle potential InterruptedExceptions. Although rare in simple scenarios like this, interruption handling becomes essential in systems where threads may be paused, stopped, or rescheduled unexpectedly. Proper handling ensures the application remains resilient and fault-tolerant during runtime.

While this program doesn’t handle external inputs or sensitive data, it is still worth discussing potential string vulnerabilities and data security in concurrent applications. Java strings are immutable and generally safe, but careless use, especially when building log errors, can inadvertently expose sensitive data or be used in injection attacks in more complex applications such as web or database-based programs. Moreover, when applications use threads to manipulate shared string data such as string buffers or formatters, developers must use thread-safe classes like StringBuffer instead of StringBuilder or apply synchronization where necessary.

All variables in this program are scoped locally within each thread’s run method, meaning they’re not shared between threads. This is good practice in concurrent programming as it prevents data races and minimizes the need for synchronization mechanisms like synchronized blocks or locks. However, if the future versions of this application were to introduce shared data like a common counter or log, it would be critical to implement proper data protection mechanisms to prevent race conditions or data corruption.

The CounterApp serves as a clean and simple demonstration of Java’s concurrency features. Through the use of Runnable, Thread, join(), and exception handling, it introduces key concepts in managing thread execution and performance. While the program is straightforward, it opens the door to deeper topics like synchronization, race conditions, exception reliability, and secure string handling. Understanding these foundational concepts is essential for building robust, high-performance, and secure Java applications in multi-threaded environments.

<https://github.com/Brandymm/CounterApp.git>

**References**

baeldung. (2018, May 5). *Baeldung*. Baeldung. <https://www.baeldung.com/java-concurrency>

Mois, M. (2015, September 16). *Introduction to Threads and Concurrency*. Java Code Geeks. <https://www.javacodegeeks.com/2015/09/introduction-to-threads-and-concurrency.html>

‌

‌