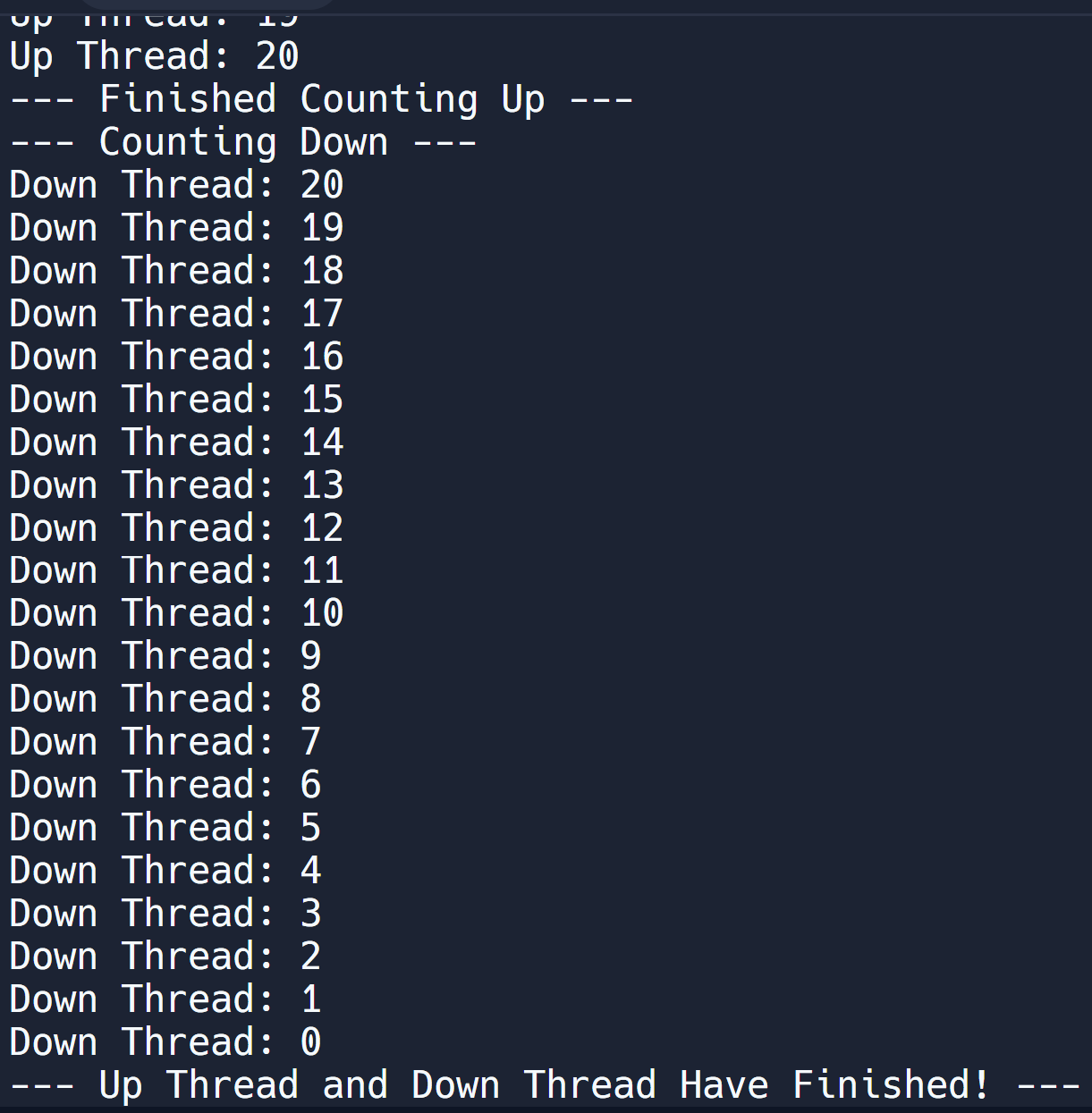
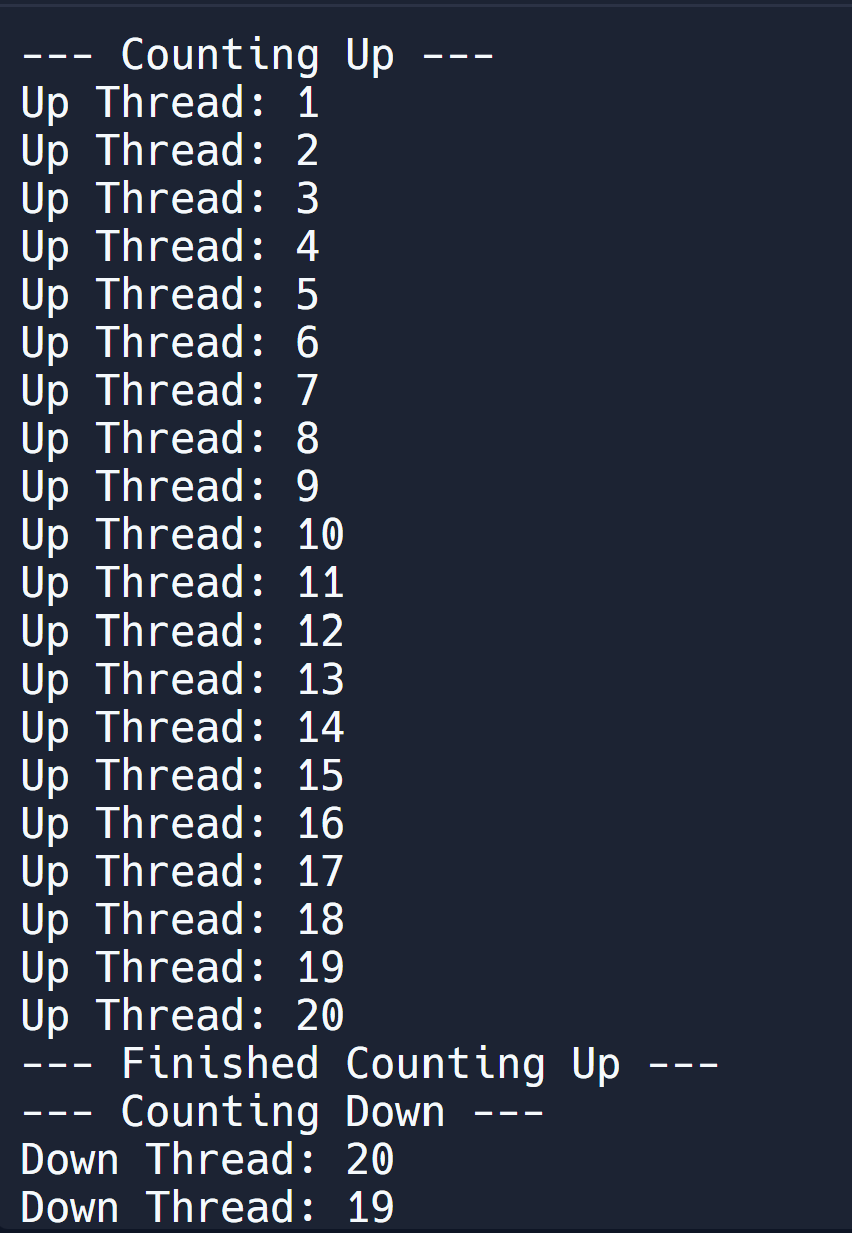
Brian Smith

CSC450-1 Module 7 PM

Reginald Haseltine

06/08/2025

**Part 1: Screenshots**



**Part 2: Code**

public class Threads {

private static int counter = 0;

private static final Object lock = new Object();

private static boolean firstThreadFinished = false;

public static void countUp() {

System.out.println("--- Counting Up ---");

for (int i = 1; i <= 20; i++) {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.err.println("CountUp thread interrupted: " + e.getMessage());

return;

}

synchronized (lock) {

counter = i;

System.out.println("Up Thread: " + counter);

}

}

synchronized (lock) {

firstThreadFinished = true;

System.out.println("--- Finished Counting Up ---");

lock.notifyAll();

}

}

public static void countDown() {

synchronized (lock) {

while (!firstThreadFinished) {

try {

lock.wait();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.err.println("CountDown thread interrupted while waiting: " + e.getMessage());

return;

}

}

}

System.out.println("--- Counting Down ---");

synchronized (lock) {

while (counter >= 0) {

System.out.println("Down Thread: " + counter);

counter--;

if (counter >= 0) {

try {

Thread.sleep(150);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.err.println("CountDown thread interrupted during countdown: " + e.getMessage());

return;

}

}

}

}

System.out.println("--- Up Thread and Down Thread Have Finished! ---");

}

public static void main(String[] args) {

Thread upThread = new Thread(new Runnable() {

@Override

public void run() {

countUp();

}

});

Thread downThread = new Thread(new Runnable() {

@Override

public void run() {

countDown();

}

});

upThread.start();

downThread.start();

try {

upThread.join();

downThread.join();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.err.println("Main thread interrupted: " + e.getMessage());

}

}

}

**Part 3: Impacts**

*Performance issues with concurrency:*

* Contention: Contention arises when locked objects are being called by multiple threads. Although my example only has two, this can lead to bigger issues when implemented in a larger application. A way to mitigate this is to synchronize only the most important sections of code and make those synchronizations as short as possible.
* notifyAll(): In this instance, this only wakes up my downThread and causes it to wait on the locked object. But, as mentioned above, if this were implemented in a larger application, this could lead to issues. Waking up all threads leads to increased contention and unnecessary CPU cycles.
* Thread.sleep(): Although the sleep timer is requested for a specific time, 150ms, this may result in an increased wait time due to system load. Because of this, artificial pauses can be inadvertently implemented, resulting in an inefficient use of work and waiting times.

*Vulnerabilities exhibited with use of strings:*

* Due to the simplicity of this application, traditional security pitfalls such as cross-site scripting, SQL injections, and command injections. There are no inputs (command injections) or databases (SQL injections) that would introduce vulnerabilities to the application.

*Security of the data types exhibited:*

* Race Conditions: In the counter, these can occur when synchornization isn’t implemented properly. If the counter is unprotected and being utilized by two or more threads, the ending total can be wrong. This result has a potential of integer overflow or integer underflow, where the counter exceedes its bounds and ends up negative or positive.
* Object Locks: Monitors the synchronization. If used incorrectly, deadlocks can be introduced resulting in endless waits and a denial of service.
* String Literals: Once their hardcoded content is created, they cannot be changed. Because these are used for printing, there is little to worry about.