**Portfolio Project**

Md Abdul Wahid Emran

Colorado State University Global

CSC450: PROGRAMMING III

Reginald Haseltine

Due Date: 02/11/2024

Below is the screenshots of Java program which demonstrates two classes, CountUp and CountDown, each implementing the Runnable interface to define tasks for counting up to 20 and counting down from 20 to 0, respectively.

**CountUp.java**

A screenshot of a computer

Description automatically generated

**CountDown.java**

A screenshot of a computer

Description automatically generated

**ConcurrencyDemo.java**

A screenshot of a computer

Description automatically generated

**Program output**

A screenshot of a computer

Description automatically generated

**Github Screenshot:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Below is a detailed analysis of the relevant ideas that might have an effect on the application:

**Concurrency's impact on performance:** By making use of the system's multiple cores or processors and cutting down on CPU idle time, concurrency can enhance application performance. Concurrency, however, can also add overhead and complexity in the form of managing thread creation and synchronization, resolving exceptions and mistakes, and synchronizing and coordinating shared data. Depending on the quantity and size of the jobs, the volume and frequency of data sharing, and the effectiveness and quality of the code, these variables may have an impact on the application's speed, scalability, and dependability.

**Vulnerabilities in how strings are used:** In Java, strings are immutable objects, which means that once they are generated, they cannot be changed. When utilizing strings in concurrent applications, this can lead to a number of issues, including the needless duplication of strings, memory and garbage collection resource wastage, and performance degradation.To avoid these problems, it is recommended to use mutable string classes, such as StringBuilder or StringBuffer, which can be modified without creating new objects. However, these classes are not thread-safe, which means that they can cause data inconsistency or corruption if accessed by multiple threads at the same time. To ensure thread safety, it is necessary to use synchronization mechanisms, such as the synchronized keyword or the Lock interface, which can lock access to the shared data and prevent concurrent modifications. However, synchronization can also introduce some drawbacks, such as reducing concurrency, increasing complexity, and causing deadlocks or livelocks. Therefore, it is important to use synchronization carefully and wisely and to minimize the scope and duration of the locks.

**Security of the data types exhibited:** The data types exhibited in the application are primitive types, such as int, and reference types, such as String and Thread. Primitive types are stored in the stack memory, which is private and local to each thread, and therefore they are not shared among threads. Reference types are stored in the heap memory, which is shared and global to all threads, and therefore they can be accessed and modified by multiple threads. This can cause security issues, such as data leakage, data tampering, or data loss, if the shared data is not protected or isolated properly. To ensure the security of the data types, it is necessary to use encapsulation and abstraction techniques, such as making the data fields private and providing public methods to access and manipulate them. It is also necessary to use concurrency control techniques, such as making the data fields final, volatile, or atomic, or using concurrent collections, such as ConcurrentHashMap or CopyOnWriteArrayList, which can provide thread safety and data consistency without explicit synchronization.