A Java Counting Program that Uses Threads Sequentially

Stephan Peters

Colorado State University Global

CSC450-1 23WD: Programming III

Reginald Haseltine

5 April 2024

A Java Counting Program that Uses Threads Sequentially

This document shows a Java application I created that exhibits concurrency concepts. In this application, I create two threads that will act as counters. The first thread counts up to 20. Once the first thread reaches 20, then the second thread counts down to 0. After both threads are completed, a third thread starts that shows the elapsed time for the threads to complete, this is used. Source Code shows the object class I use to generate the object. Note, this class has a dependency, `edu.princeton.cs.algs4.Stopwatch`, used for benchmarking to compare the performance of the Java application to an equivalent C++ application.

Source Code 1

The Count class: Count.java.

```
import java.util.concurrent.CountDownLatch;
import edu.princeton.cs.algs4.Stopwatch;

/**
 * A class that represents a count variable
 * that stores minimum and maximum, and increment values.
 * Default values for minimum and increment are 1
 */
public class Count {
    Stopwatch stopwatch;
    CountDownLatch latch = new CountDownLatch(1);
    private int min = 1;
    private int increment = 1;
    private final int MAX;

    /**
     * Represents a count variable
     * that stores minimum and maximum, and increment values.
     * @param max minimum value
     */
    public Count(int max) {
        this.stopwatch = new Stopwatch();
        this.MAX = max;
    }
```

```java
    /**
     * Represents a count variable
     * that stores minimum and maximum, and increment values.
     * @param min minimum value
     * @param max minimum value
     */
    public Count(int min, int max) {
        this.stopwatch = new Stopwatch();
        this.min = min;
        this.MAX = max;
    }

    /**
     * Represents a count variable
     * that stores minimum and maximum, and increment values.
     * @param min minimum value
     * @param max minimum value
     * @param increment increment value
     */
    public Count(int min, int max, int increment) {
        this.stopwatch = new Stopwatch();
        this.min = min;
        this.MAX = max;
        this.increment = increment;
    }

    /**
     * Prints the elapsed CPU time (in milliseconds)
     * since the Count object was created.
     */
    public void getElapsedTime() {
        synchronized (this) {
            try {
                latch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.printf("\nDuration using the Java method: %.1f ms\n",
                                  stopwatch.elapsedTime() * 1000);
        }
    }

    /**
     * Counts up from minimum to maximum by the increment value.
     * By default, does not wait during concurrent threading.
     */
    public void countUp() {
        synchronized (this) {
            for (int i = min; i <= MAX; i += increment) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
```

```java
    /**
     * Counts up from minimum to maximum by the increment value.
     * Notifies other threads when the count is done.
     * @param wait whether to wait for the other threads to finish.
     */
    public void countUp(boolean wait) {
        if (wait) {
            try {
                latch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        countUp();
        latch.countDown();
    }

    /**
     * Counts down from maximum to minimum by the increment value.
     * By default, does not wait during concurrent threading.
     */
    public void countDown() {
        synchronized (this) {
            for (int i = MAX; i >= min; i -= increment) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }

    /**
     * Counts down from maximum to minimum by the increment value.
     * Notifies other threads when the count is done.
     * @param wait whether to wait for the other threads to finish.
     */
    public void countDown(boolean wait) {
        if (wait) {
            try {
                latch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        countDown();
        latch.countDown();
    }
}
```

The above class allows the object to be created and allows the parameters stored in the object to be used to count up or down. It creates an `edu.princeton.cs.algs4.Stopwatch` object as soon as the Count object is created that will act as a timer.

The driver class (TwentyTwenty.java) creates a Count object, then creates and runs the three threads sequentially. It is shown in Source Code 2.

Source Code 2

The driver program: Count.cpp.

```
/**
 * A Java application that will exhibit concurrency concepts.
 * The application creates two threads that will act as counters.
 * One thread counts up to 20.
 * Once thread one reaches 20, then a second thread counts down to 0.
 * Includes a timer thread that will print the elapsed time
 * for performance comparison to a C++ implementation.
 */
public class TwentyTwenty {
    static Count count = new Count(0,20);

    public static void main(String[] args) {
        Thread countUp = new Thread(() -> count.countUp(false));
        Thread countDown = new Thread(() -> count.countDown(true));
        // This additional thread is used for benchmarking.
        Thread timer = new Thread(() -> count.getElapsedTime());
        countUp.start();
        countDown.start();
        try {
            countDown.join();
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        timer.start();
        try {
            timer.join();
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}
```

The full `edu.princeton.cs.algs4.Stopwatch` dependency package may be downloaded directly from GitHub here:

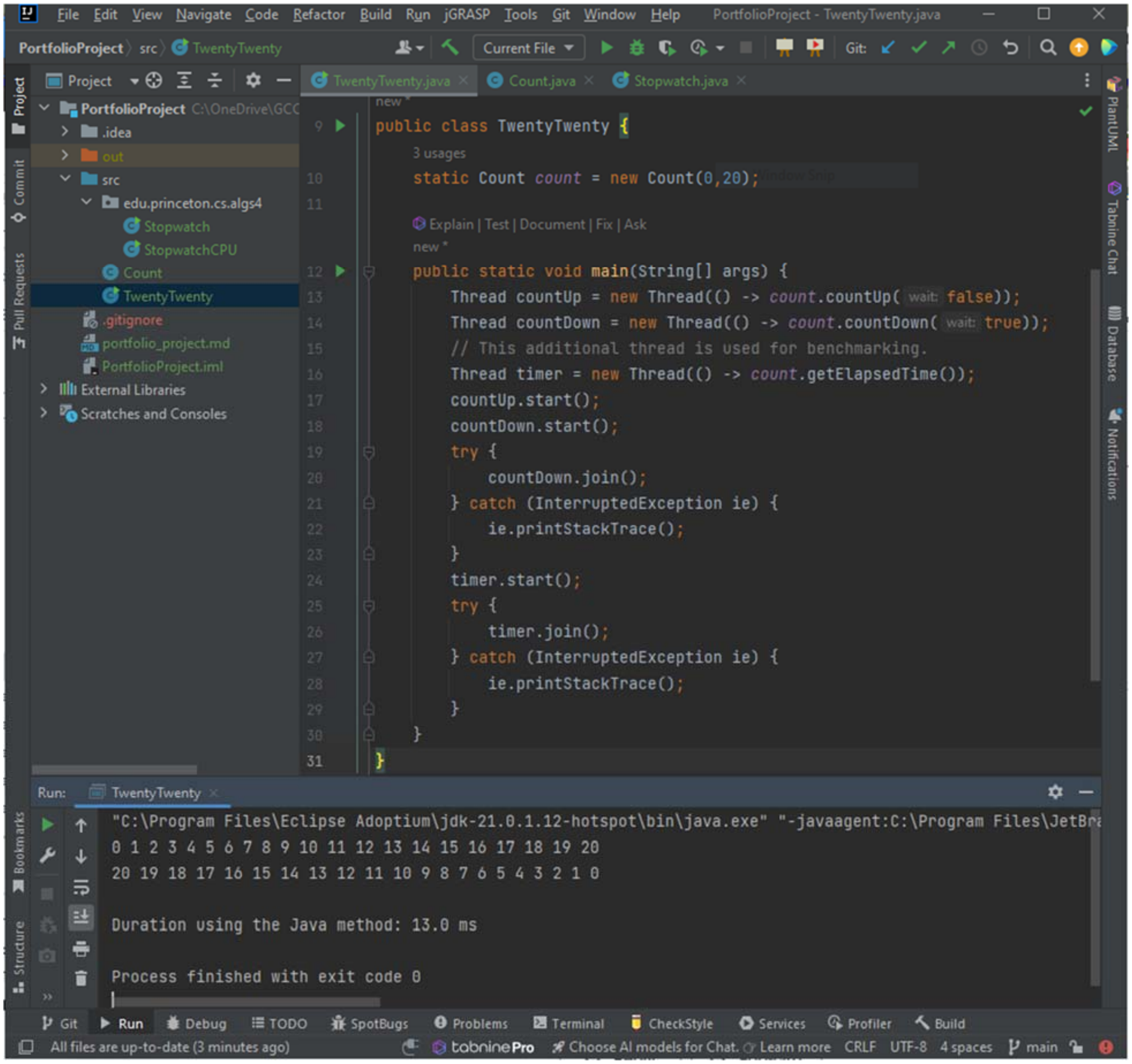
https://github.com/kevin-wayne/algs4/archive/refs/heads/master.zip

The relevant classes from the package are included in the submission.

Figure 1 shows the program running. The Count object is created, then three threads are created. The first thread runs and counts from 0 to 20. The second thread waits for the completion of the first thread using join() and counts down from 20 to 0. The timer thread waits for the completion of the second thread, then gives the elapsed "wall clock" time in milliseconds.

Figure 1

Running TwentyTwenty.java



GitHub Repository

In addition to the solutions above, I was tasked to create a GitHub repository for the project. This repository is located at https://github.com/speters33w/CSUGlobal_CSC450/tree/main/PortfolioProject

Figure 2 shows a screenshot of the main page of this repository.

Figure 2

Image of the main page of my CSC 450 Portfolio Project Milestone GitHub repository