Parallel Processes

January 27th & 28th 2020

Notes last updated: January 22nd 2020, at 6.12pm

1 The Code

This week's code bundle contains two packages: `counter` and `ui`. Only the `counter` package is essential to this week's exercises. The `ui` package merely provides a graphical interface for the code in the `counter` package.

1.1 The `counter` package

The counter package contains the following classes and interfaces:

- `Counter.java`: This is a "shared counter" `Thread` class. This class is incomplete (see below), but when complete a `Counter` thread will (attempt to) count from a start value to an end value. All `Counter` threads share the same internal counter, so it is possible that two or more `Counter` threads running concurrently might compete to change the value of the internal counter in conflicting directions. The `Counter` class also contains static methods that can switch tracing of `Counter`s on or off, and static methods that can be used to affect the speed at which `Counter`s work. All of these methods are inherited from...

- `AbstractCounter`: This class defines all of the methods needed by `Counter`, and also manages a lot of internal administration for counters, that is not directly relevant to completing the implementation of the `Counter` class. You can probably safely ignore this class. The methods from this class that are relevant to the exercises are inherited "explicitly" in the `Counter` class — e.g.:

  ```
  public void startCount() {
     super.startCount();
  }
  ```

  and

```java
public static void traceOn() {
   AbstractCounter.traceOn();
}
```

Neither of these method definitions does anything that isn't already done by normal inheritance, but they do improve the "visibility" of these methods for developers.

- `CounterException`: This is used for errors in `Counters`, usually in the initial values used to construct a `Counter` — e.g. trying to construct a `Counter` that attempts to count from 0 to 5 in steps of -1. It is also used to report errors in trying to set the delays used to slow down `Counters`.

- `ThreadSet`: This interface defines a set of `Threads`, and requires any implementation to implement `startSet()` and `joinSet()` methods that will, respectively:

  - start up all of the `Threads` in the set concurrently;
  - and wait for all of the threads in the set to finish.

- `ThreadHashSet`: This is an incomplete (see below) implementation of the `ThreadSet` interface.

- `Main.java`: This class contains a main method that demonstrates how `Counters` and `ThreadSets` can be used. It also switches on tracing of `Counters` so that their behaviour can be observed.

  The `ui` package also contains a `Main` class, which also demonstrates the behaviour of `Counters` through animation in the interface. When trying to observe `Counters`' behaviour in order to answer the questions below I would recommend using both means of doing so, as the observed behaviour may differ in both packages (the extra time taken by the graphical interface affecting effective time slices for the `Counters`).

1.2 The `ui` package

This package implements a graphical interface for the `counter` package.

You do not *have* to use this package in order to complete the exercises. This can be done using solely the code in the `counter` package. This package is provided *solely* in order to provide a basic visual presentation of counters. There should be no need for you to edit any code in this package.

Counter configurations (sets of counters) can be constructed, loaded and saved using the Configuration menu. A basic configuration file, `config.cnt`, should be available in this week's code bundle. A configuration defines a "run set" of counters that will be loaded ready to run, and possibly also a set of predefined counters that are not currently in the run set, but that can easily be loaded into it (see below).

A sample configuration file, `config.cnt`, is provided, which defines a configuration consisting of a run set with two counters, one of which counts from 5 to 10, in steps of one, and the other of which counts from 5 to 0, in steps of minus one. The predefined counters set of this configuration also contains these counters, plus two more counters, which count from 0 to 10, and from 10 to 0, in step of 1 and -1.

The current configuration can be saved to a configuration file, using the "Save configuration" and "Save configuration as" options. Once a configuration file has been saved it can be loaded into the interface using the "Load configuration" or "Reload last configuration" option.

Configurations can also be edited "by hand" by using the "Add counter", or "Remove counter" options. Under "Add counter" you are given the choice of using a predefined counters, if these are available in the current configuration, or of creating a new counter. When a new counter is defined, you are given the option of either adding it to the current run set, or to the set of predefined counters.

In the panel used to create a new counter the values for the step value, and for the counter's name will be entered automatically. These automatic values can easily be overridden by editing the relevant fields in the panel.

In the "Remove counter" menu, "Remove all" will remove all counters (i.e. clear the run set, or the set of predefined counters).

The "Show . . . " options will simply list all the counters in the current run set or predefined counters set.

The Trace menu allows you to switch tracing on or off (default is on), and to save the trace output to a file.

The "Run" button will run the current run set, and the "Stop" button can be used to signal that all counters should stop, whether or not they have reached their limiting value. The current value of the count is shown in the value bar.

Counters are shown to the left and right of the trace window, with decrementing counters on the left, and incrementing counters on the right. Counters that have not yet started are shown in orange. Counters in yellow are active, but not actually accessing the shared count. When a counter does access the shared count (to set, change, or check the value) it will be shown in green. A counter that has finished its run will be grey, except if it was stopped by the "Stop" button, in which case it will be red.

Note that once a run set has been run, it will need to be reloaded before it can be run again.

The speed slider can be used to control the speed of execution. Counters will not react *immediately* to any change in the speed slider. Each counter will have to complete its current pause (if it is pausing) before the speed change takes effect.

The code in this package should definitely *not* be considered as an example of good coding practice. Please report any bugs, or any suggestions for improvements to me.

2 Programming Exercises

- **Model question.** The `run()` method in the `Counter` class is currently just a stub. Implement the `run()` method so that when a `Counter` thread is run it will start a **while** loop to run through all the values of the counter. Note: this is *easy!* It does not require any knowledge or experience of concurrent programming. Think of it as an exercise suited to an introductory course in Java progamming.

  Have a look at the last few methods defined in the `Counter` class. Using these to implement the loop should then be trivial.

- The `ThreadHashSet` class claims to implement the `ThreadSet` interface, but the `startSet()` and `joinSet()` methods demanded by the interface are also just stubs. Implement these methods. The `startSet()` method should start all the `Threads` in the set running concurrenlty, and the `joinSet()` method should wait for them all to finish. This is slightly more difficult, as you need to manage starting up the `Counter` threads, and then waiting for them to stop. See the lecture notes for information on how to do this. If necessary, use "for each" loops to iterate through all the `Threads` in the set:

  ```
  for (Thread thread:  this) {
      ...
  }
  ```

3 Demo Code

The `Main` class contains a `main` method demonstrating the use of `Counters` and `ThreadSets`. In this method a `ThreadSet` is populated with two `Counters`, one that tries to count from 5 to 10, and another that tries to count from 5 to 0. Tracing is switched on, so that when the code is run the behaviour of the `Counters` can be observed (if you have correctly implemeted the `run()` and `runSet()` metods).

Try running the `main` method a few times and observe the `Counters`' behaviour. Do you see something like:

```
up has started: 5
down has started: 5
up has stepped: 6
down has stepped: 5
up has stepped: 6
down has stepped: 5
down has stepped: 4
up has stepped: 5
down has stepped: 4
```

```
up has stepped: 5
down has stepped: 4
up has stepped: 5
up has stepped: 6
up has stepped: 7
```

. . . , or is:

```
up has started: 5
up has stepped: 6
up has stepped: 7
up has stepped: 8
up has stepped: 9
up has stepped: 10
up has finished: 10
down has started: 5
down has stepped: 4
down has stepped: 3
down has stepped: 2
down has stepped: 1
down has stepped: 0
down has finished: 0
```

more like your output?

If the latter, there is almost certainly a problem with your code so that the counters are not running concurrently. Things that you can check:

- Do you call the counters' `start()` method (which will start them up as concurrent threads), or do you call their `run()` method (which will run each counter as a sequential process)?
- Do you start *all* counters running *before* you wait for any of them to finish?

Once your counters appear to properly run concurrently try editing the `main` method to change the `Counter` delay to low and high delays, and try running the code again. You may observe a difference in behaviour. If you do, what is this difference, and why do you think it occurs?

4 Logbook Exercise

Edit the `main` method so that the `Counters` now try to count from 0 to 10, and from 10 to 0, in steps of ± 1. Make sure that the `Counters` trace their behaviour.

Run this revised method a number of times, and answer the following questions. Make sure that you explain your answers.

Note: these questions must be answered for the $(0 \rightarrow 10, 10 \rightarrow 0)$ counter pair, *not* for the $(5 \rightarrow 10, 5 \rightarrow 0)$ counter pair.

1. Will the test always terminate? I.e. is it certain that no matter how often you were to run the test it would always end in a finite length of time?
2. What is the shortest possible output for the test, in terms of the number of lines output?
3. What is the largest possible value that the count can reach when the test is run?
4. What is the lowest possible value that the count can reach when the test is run?