Lab: Objects and Classes
CSC 207, “Algorithms and Object-Oriented Design”
Department of Computer Science
Grinnell College
February 4, 2021

Let’s start building up an inventory of simple classes that might be useful components for future work.

1. (2 points) One of the libraries that instructors in the current CSC 151 course have been supplying to their students implements an abstract data type called `counter`. You can think of a counter as a simulation of those old-fashioned hand-held clickers that people used to use to count attendees at a county fair as they pass through the turnstile at the entrance gate or to tally cars as they cross a bridge. Such a clicker is basically a numerical display with a tally button that increments the number displayed and a reset button to reset that number to zero.

The implementation of this data type in CSC 151 provides a constructor (`counter-new`), an accessor (`counter-get`), and restricted mutation procedures for incrementing the internal tally and resetting it to zero (`counter-increment` and `counter-reset`).

Reimplement this ADT as a Java class, `Counter`, and test the followings after instantiating `Counter` in the `main` method: 1) Increment the counter and display its value, 2) Increment the counter multiple times, reset it, increment the counter, and then display its value.

Remember to name your methods using Java naming rules that was discussed in the class. Also, create a tester class file to test the implementation of the methods.

2. (1 point) In the implementation of counters in CSC 151, each counter produced by the constructor actually carries around a name which is specified as a string argument to the constructor. This name is supposed to be immutable, since none of the ADT procedures modify it, and it is used only by a `counter-print` procedure that displays the internal state of the counter.

Add a private `name` field to your `Counter` class. Add a `String` argument to the constructor that you wrote for the preceding exercise, and makes it the name of the counter. Add a `toString` method that constructs and returns a `String` consisting the counter’s name, a colon, a space, and the current tally. Test this feature by creating an instance of a counter, incrementing its value, and calling the `toString` method to display the string representation of the counter.

3. A slightly more sophisticated device is the `averager`, which allows the user to add any number, not just 1, to a running total that it maintains, and also keeps track of the number of addends that it has received since it was created or last reset. The averager can divide its running total by its running tally of addends to report the arithmetic mean of all of those addends.

For a minimal implementation, you’ll need a constructor, a mutator that takes a numerical argument and adds it to the running total (simultaneously incrementing the addend tally behind the scenes), a mutator that resets both the running total and the addend tally to 0, and an accessor that gives you the average of the addends.

The arithmetic mean is undefined when the addend tally is 0. In this special case, the accessor method should return 0 after printing a brief error report to `System.err`, which is a `PrintStream` analogous to C’s `stderr` file descriptor.

Implement and test an `Averager` class by 1) computing and displaying the arithmetic mean of five positive and negative numbers, and 2) reporting the error as explained above.

4. Other potentially useful features for averagers include separate accessors for the running total and the addend tally, `toString` and `equals` methods, accessors for the maximum and minimum values among the addends seen so far, and a method that takes another `Averager` as argument and combines its running total and addend tally with the running total and addend tally inside
this. Note that the `equals` method takes `Object` as its parameter. Thus, you need to do a type conversion. What should you check before doing the type conversion?