Lab 7: Recurring Recursion

CSCI 1101B – Spring 2015

Due: April 21, 10 pm

Objective: To gain experience using recursion.

This lab is divided into two parts (but you will complete both parts in the same week). In part 1, you will write two nongraphical, recursive methods involving `ints` and `Strings`. In part 2, you will draw an interesting graphical structure recursively.

Part 1: Warmup

In this part, you will write two recursive methods in the `TextEvents` class (*not* the `Events` class, which will be used in part 2). This class already contains code in the `begin` method that tests the methods you should write – do not change the existing test code, although you may add additional code at the end of the `begin` method if you wish.

Your method implementations **must** be recursive! You will receive no credit for iterative solutions (i.e., those using loops).

Number of Digits

Write a recursive method called `numDigits` that accepts as input an integer `n` and returns the number of digits in that number. You may assume that `n` is greater than zero. Note that `n` is *n*ot a `String`, and are you *not* allowed to turn it into a `String`.

Palindromes

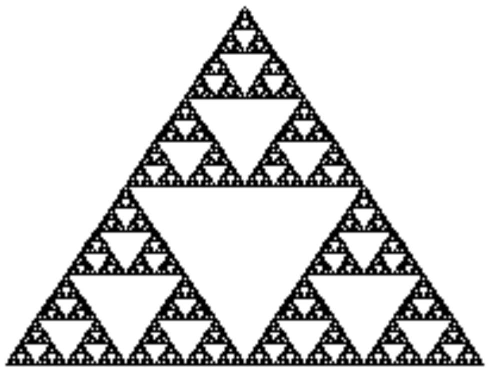
During our discussion of `Strings`, we looked at an example program in which we wrote a method called *isPalindrome* that accepted a `String` as input and returned whether that `String` was a palindrome (i.e., a word that is the same forwards or backwards). This method was an example of an **iterative** solution, as it relied on a `while` loop to solve the problem by iterating through the characters of the string (in reverse).

Write a second version of the `isPalindrome` method that is **recursive** rather than iterative (i.e., without using a loop). You may assume that the input string is a single word (i.e., no spaces) and that it is all lowercase.

NOTE: You cannot use any `String` methods except `length`, `substring`, `charAt`, and `equals`. You may not necessarily need all of these methods, but you may not use any other `String` methods besides these.

Part 2: Sierpinski's Gasket[1]

Interesting geometric patterns can be produced by drawing smaller and smaller copies of a shape inside an initial copy of that shape. One such pattern is called Sierpinski's Gasket, which you will draw in part 2 of this lab:



Because this pattern is going to be constructed recursively, you will need to design a class with a recursive constructor (and a recursive `move` method). We will call this the `ComplexTriangle` class. One way to think about constructing this recursively is to draw the outermost triangle (given its vertices) and then construct three smaller `ComplexTriangle`s inside that triangle: one at the top, one at the lower left, and one at the lower right. Thus, the `ComplexTriangle` class will need instance variables for the lines composing the outer triangle as well as instance variables for the three `ComplexTriangle`s inside. Besides the constructor, the only other methods that will be needed are a `move` method and a helper method `getMidpoint` (described below).

The constructor will take parameters for the vertices of the triangle (in the order `left`, `right`, and `top`) and the drawing canvas. It should first draw lines between the vertices to form a triangle. Then, if the bottom edge of this triangle is long enough (say, with length strictly greater than 5 pixels), it should construct three more `ComplexTriangle`s inside this triangle. To create the smaller triangles, you should find the midpoints of each of the sides of the triangle just drawn and then create a `ComplexTriangle` in the top, lower-left, and lower-right portions of the triangle (leaving the middle blank). If the edge of the outer triangle is not long enough, then the constructor should not make the three `ComplexTriangle`s.

To avoid writing duplicate code to find a midpoint three times (for the three edges of the outer triangle), you should write a helper method called `getMidpoint` that is sent a `Line` and returns the midpoint of that line (i.e., a `Location`). To help in writing this method, you can use the `getStart` and `getEnd` methods of the `Line` class to get the start and end `Locations` of the line, respectively.

[1] Adapted from a lab provided with *Java: An Eventful Approach*, K. Bruce, A. Danyluk, and T. Murtagh

You should also write a `move` method that moves the lines forming the triangle as well as the contained `ComplexTriangle`s.

I have provided you with an `Events` class and the bare skeleton of a `ComplexTriangle` class in the starter project. The `begin` method constructs a new `ComplexTriangle` object. The `onMousePress` and `onMouseDrag` methods use the `move` method that you will write in your `ComplexTriangle` class to drag the `ComplexTriangle` around, even when the mouse is not inside the `ComplexTriangle` (because there's no straightforward way to check whether the user clicked inside the `ComplexTriangle`). You should not need to modify the `Events` class at all (though feel free to change the constants defining the triangle while experimenting with your program).

Tips

- To start out, it is often a good idea to do a simpler version of the problem. For example, first have the constructor of `ComplexTriangle` just draw a simple triangle out of lines. Then try drawing only the `ComplexTriangle` in the top part of the triangle. Then add the lower right `ComplexTriangle`, then finally add the triangle in the lower left portion.
- You will find the `distanceTo` method defined in the `Location` class useful in determining whether to draw the inner triangles. If you have a `Location` object, you can call this method along with a `Location` object argument, and the result will be a double indicating the distance between the two locations; for example:

  ```
  Location x = new Location(20, 50);
  Location y = new Location(65, 150);
  double distanceXY = x.distanceTo(y);
  ```
- If you are not careful in writing your program, you may get a `StackOverflowError` when you run the program. This will occur if your program continues to construct `ComplexTriangle`s without ever terminating. You should avoid this by making sure your `ComplexTriangle` constructor stops creating `ComplexTriangle`s at some point.

Submitting Your Work

Submit your compressed lab folder (named using your last name) in the usual way on Blackboard. Your project should contain all three classes – `TextEvents` (from part 1), `Events` (from part 2), and `ComplexTriangle` (from part 2). Remember to put your name at the top of your files and fill in all missing documentation (e.g., method and program summaries).