2/4/14

3. Homework

Programming portion (problems 1(a)-(d)) due **Tuesday 2/11/14** at 11:55pm on Blackboard.
Written portion (problems 1(e), 1(f), and 2) due **Wednesday 2/12/14** at the beginning of class.

Please zip the (Eclipse) project directory for this homework, and use the following naming convention for the name of the project (and directory): `lastName_firstName_hw3`. **In order to receive any credit for the programming portions, you are required to thoroughly comment your code.**

1. **Flood Fill (11 points)**
   Here we will use stacks and queues to implement the *"flood fill"* algorithm that is commonly used in computer graphics. For this homework we will use a GUI interface for the first time. You will not need to modify the GUI portion of the code, other than to change the image being viewed. Please download the template code that contains the following files `ImageViewer.java`, `FillComponent.java`, and `Point.java` in the `src` folder, as well as various image files in the `inputPictures` folder.

   (a) (0 points) The class `ImageViewer` contains the `main()` method in which it launches the viewing application. This class also loads the image by creating an instance of the `FillComponent` class. To change the image being loaded, you need only to change the argument to the `FillComponent` constructor, by providing the filename to the image. The `ImageViewer` class also monitors mouse activity and launches the method `mouseClicked` when the mouse is clicked on a part of the loaded image. To get familiar with this framework, load the `shapes.png` test image by providing the correct path and run `ImageViewer`.

   (b) (2 points) As a warmup, implement the `sillyFill` method in `FillComponent`. Make sure to call the `sillyFill` method from `mouseClicked` method in `ImageViewer`. In `sillyFill`, try to change the color of a couple of pixels (it's hard to see the color of a single pixel changed): You can draw a "line segment" of 20 pixels: (x, y), $(x + 1, y)$, ..., $(x + 19, y)$. Or you could draw a square of 20 times 20 pixels. Note that to set a pixel in the image to a particular color, you must call the `setRGB` method of the `BufferedImage` member variable `bi` from within `FillComponent`. Then, you must call `paintImmediately()` to update the screen image.

   (c) (2 points) Modify the dynamic `Stack` and the dynamic `Queue` classes given in class so that they both can hold generic types, and they both implement a method called `isEmpty()` which returns a boolean value indicating whether the data structure is empty. Below you will use a queue of `Point` objects, `Queue<Point>`, or a stack of `Point` objects, `Stack<Point>`.

FLIP OVER TO BACK PAGE ⟹

(d) (4 points) Implement *flood fill* functionality in the `floodFill()` method of `FillComponent`. As you may know, the flood fill feature of a graphics program "fills" a contiguous area of one color with a new color starting at a chosen pixel. Suppose the integer RGB code for the starting pixel is `oldColor`, then flood fill works its way outward from the starting pixel and paints pixels in the `newColor` in a contiguous manner (in particular, it only recolors pixels of color `oldColor`). More concretely, the flood fill algorithm works as follows: First, we store the starting pixel as a `Point` in either a queue or a stack of `Point` objects, and then we repeat the following until our data structure is empty: We retrieve a stored `Point`, add all of its neighbors with color `oldColor` to our data structure, and update the color of the retrieved pixel to `newColor`.

Note that to test this method, you must uncomment the invocation of `sillyFill` in `mouseClicked` in `ImageViewer`.

(e) (1 point) Use the test image `shapes.png` to ensure that your flood fill algorithm works correctly when a stack or a queue is used. What differences do you observe in the way the fill progresses, depending on whether a stack of queue is used? Can you explain why? *(Note that floodfill is actually a graph traversal with pixels as vertices and edges between neighboring vertices. Which of the flood fills reminds you more of depth-first search, and which of breadth-first search?)*

(f) (2 points) We can even solve mazes using the flood fill algorithm: Use your flood fill implementation to decide whether each of the mazes in the provided folder are solvable and report your findings. Use the `.png` files as input to your code above; I have also provided `.pdf` files if you'd like to work through these inputs by hand. Starting and ending points are indicated by dots or the characters "S" and "E".

2. **Shape Hierarchy (9 points)**
Consider the the file `hw3-shapeHierarchy` containing the files `Tester.java, Point.java, Shape.java, Rectangle.java, Circle.java`.

(a) (6 points) Draw the state of the memory at the end of the `main` method in `Tester.java`. Be aware of the difference between primitive types and references. For each object, make sure to show the values of all attributes including the attributes from the super class.

(b) (1 point) At the end of the `main` method in `Tester.java`, what does `System.out.println(rec.getID())` print? Why is it this value?

(c) (1 point) At the end of the `main` method in `Tester.java`, what does `System.out.println(c.getID())` print? (This is a trick question...) Explain your answer.

(d) (1 point) At the end of the `main` method in `Tester.java`, what does `System.out.println(c)` print? Why does the `Circle c` have access to the `getX()` and `getY()` methods when they are not implemented in `Circle`?