PSA 5: Polymorphism and Recursion with Abstract Shapes

Read all instructions in this document before starting any coding!

In this assignment, we will write a family of Shapes as sources for us to draw animations with. Inheritance and overriding are revisited, but we are already familiar with it. We continue the exploration of object-oriented principles with the techniques of overloading, polymorphism, and recursion. Note the changed grading breakdown scheme.

Helpful Information:

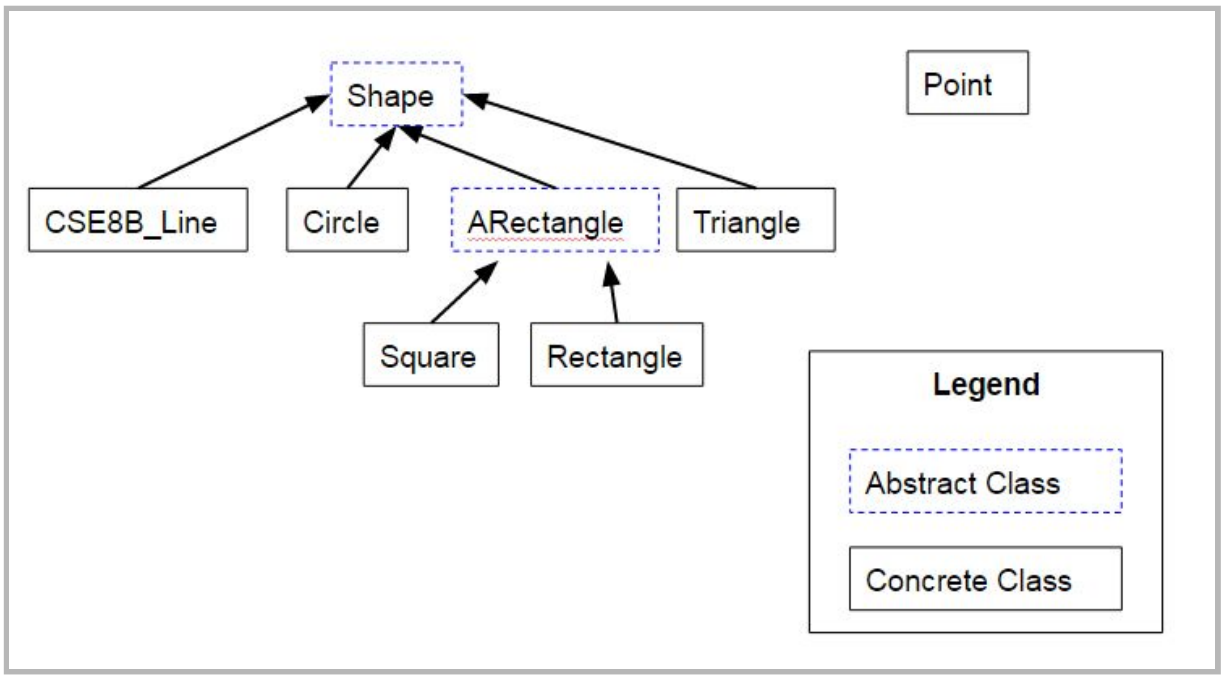
- Online Communication: Using Piazza, Opening Regrade Requests
- Getting Help from Tutor, TA, and Professor's Hours
    - Lab and Office Hours (Always refer to this calendar before posting.)
- Academic Integrity: What You Can and Can't Do in CSE 8B
- Setting Up the PSA
- How to Use Vim
- Style Guidelines
- Submitting on Vocareum and Grading Guidelines

Table of Contents:

Part 1: Abstract Shapes [60 Points]

Shape Hierarchy

Write a set of classes to implement a simple hierarchy of Shapes as follows:



Link to a more detailed UML diagram of the shape hierarchy.

Note:

- You need to implement the methods/classes in the above files. We have provided the java files in the starter code. You just have to write your code and implement the method wherever you see the 'TODO' comment
- You DO NOT have to implement the code to draw the “Mickey” or the “House” or any of the testers for this assignment.

How to Compile and Run

To compile your code, use the following and note the different commands per operating system:

For Mac, Linux, AND Ubuntu Bash on Windows:

```
> javac -cp ./objectdraw.jar:. TestMickey.java
or
> javac -cp ./objectdraw.jar:. TestHouseWithDelaysController.java
```

For Windows without Ubuntu Bash, use the Command Prompt (NOT PowerShell):

```
> javac -cp objectdraw.jar;. TestMickey.java
or
> javac -cp objectdraw.jar;. TestHouseWithDelaysController.java
```

To run your code, use the following:

```
> appletviewer TestMickey.html
or
> appletviewer TestHouseWithDelays.html
```

You can follow the same process with TestSquares, TestCSE_8BLine, TestTriangle, and TestRectangle.

IMPORTANT: Do not use the JavaFX library. If you use the library, we will dock points, maybe all of them. You will get a chance to use this library in your next PSA.

Getters and Setters / Accessors and Mutators

You will lose substantial style points if you do not use getters and setters. Essentially, they're a good programming technique that you will use in future classes. For this assignment you will be writing public getters and private setters for every class.

All accessing of private data in each class must be made through the appropriate get/accessor and set/mutator methods; do not directly access data, including in constructors. The only place where direct access is allowed is in the actual accessor/mutator methods

For example, for Rectangle.java you have two private ints called **width** and **height**. Thus, you would need the following methods:

- **getWidth()**
- **getHeight()**
- **setWidth(int width)**
- **setHeight(int height).**

If you need to assign the value 2 to the **private int width** then you would rather call **setWidth(2)** instead of **this.width = 2** and there would be code in your **setWidth** method to assign a value to width.

For example, consider the Rectangle constructor: Rectangle(int x, int y, int width, int height), if you want to assign its width to the passed in width 2, instead of using this.width = 2, call this.setWidth(2). If you want to get the width or height of the rectangle in any place, instead of using this.width or this.height to access them, call this.getWidth() and this.getHeight(), which will return the corresponding width and height respectively.

If you want to use **private int width** in a calculation or somewhere else then you would use **getWidth()** in place of **return this.width**

Overview of Methods and Objectdraw

move()

The **move()** method adjusts the shape xDelta pixels in the X coordinate and yDelta pixels in the Y coordinate. Just add these deltas to the shape's current X and Y location depending on how that shape's location is represented -- CSE8B_Line: both start and end Points; Circle: center Point; Rectangle: upperLeftCorner Point; Triangle: all 3 Points.

draw()

The first draw() method should create the appropriate objectdraw library object to draw on the *canvas* parameter. It takes three parameters: a canvas, a color, and a fill boolean. The canvas is where you will draw your shapes on, the color is the color of the shape, and the boolean parameter *fill* indicates whether the graphical object should be filled or not (for example, for Circle whether a FilledOval or a FramedOval should be created). This has no meaning in CSE8B_Line. If the Color parameter is null, use the black color.

In addition, for triangle, since there isn't a corresponding object in objectdraw library, think of it as a shape comprised of three lines and you don't need to consider the fill situation for triangle.

There are two draw() methods to be implemented for each concrete class, in which we overload one from the other. The second draw() with only a canvas parameter should draw the shape required by doing a method call to the first draw(), adding its own random Color and a fill boolean of true as arguments.

Because the **objectdraw** library already has a Line type, we name our Line type **CSE8B_Line** as to not confuse/conflict with the objectdraw library's **Line**. The **draw()** method in our CSE8B_Line will use the objectdraw library's Line type.

toString()

The toString() method is what prints out the strings you see i

n the example pictures below.

Check your formatting of the string against our examples for each object.

Objectdraw Library and Constructors

As you may have noticed, when you compile you specify the class path (that is additional classes or libraries that your program may need to compile, such as objectdraw.jar). This file contains information about the shapes or objects in the objectdraw library. The objectdraw library was created by several people in an effort to teach Java to students.

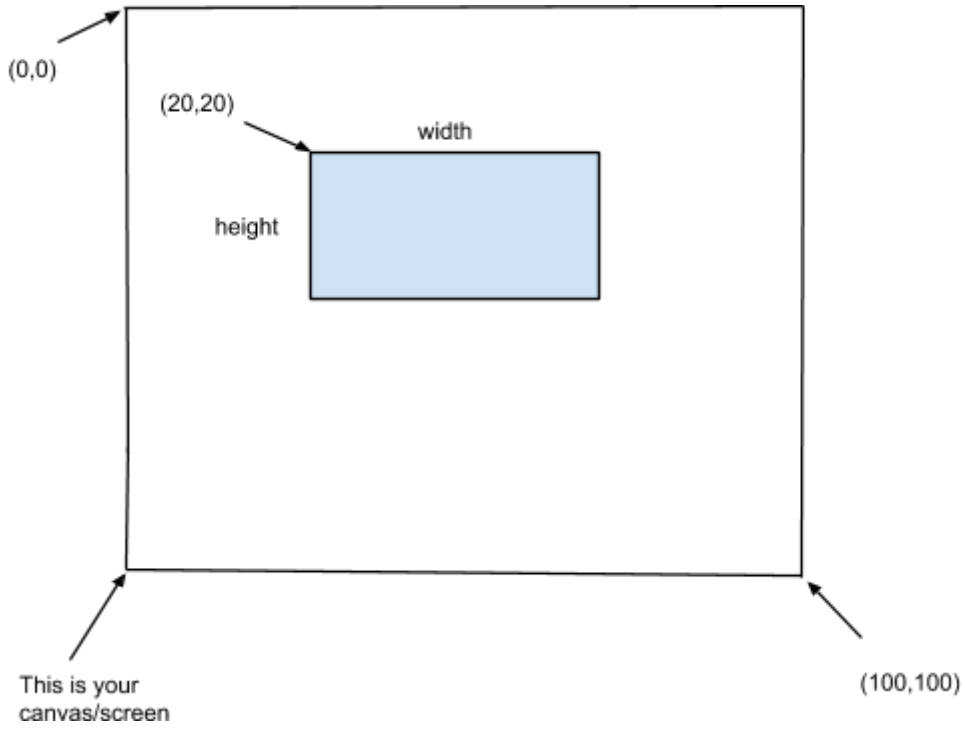
The following link is the documentation of the library which you will be using in your shapes. The API contains information on the constructors and its methods for drawable objects, including the FilledRect that you will use in Rectangle.java.

http://eventfuljava.cs.williams.edu/library/objectdrawJavadocV1.1.2/index.html

For example, to create a FilledRect object you would fill in the constructor's parameters but it would essentially look like this:

FilledRect myRectangle = new FilledRect(double x, double y, double width, double height, DrawingCanvas canvas);

The x value and y value would be calculated from the upper left point of the FilledRectangle which is considered as the origin. Width is how wide the box is and height is how tall it is.



The same concept applies for **FramedRect, FilledOval, FramedOval**, etc. Note, for the oval shapes, pretend like there's an imaginary rectangle around the oval. The x, y values for the oval constructor would be the upper left corner of the imaginary rectangle. The width of the oval would be the diameter of the oval (as we are only using ovals to draw circles). The height of the oval would be the diameter as well.

Copy constructors initialize the instance variables from an existing object of the same type to this newly created object. If the variable is a primitive data type, just copy the value of this primitive type from the existing object to this object (assignment through mutator method). If the variable is a reference to an object, create a new copy of the object this variable is referencing (by invoking that variable's copy ctor) and assign this resulting new copy of the object to this object's instance variable. This should result in a deep copy.

For example, in class Center's copy ctor,

```
public Circle( Circle c )
```

to set the center instance variable properly to a new Point based on the parameter's center Point:

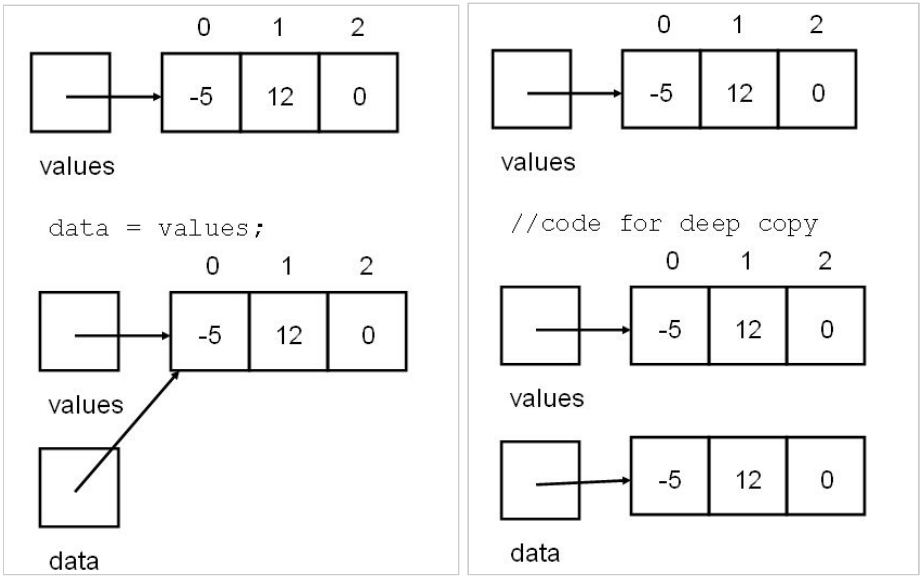
```
this.setCenter( new Point( c.getCenter() ) );
```

For a bit more information about copy constructors and deep copying, see the discussion slides. Also, thanks to the folks at UTexas, here are some diagrams.

The left diagram shows a shallow copy. values is an array. data has a copy constructor that was passed in values. As a result, rather than data pointing to a new array with the same numbers as in values, data points to the same array as values. They both refer to the same object. Now look at the right hand side. Here is an example of deep copying. data and values do not refer to the same object. They are both their own arrays. However, they have the same numbers and the same size. This is the type of copy your copy constructors should do.

How could the concept of deep copy can affect this program? For example, if you create a circle A first and make a copy(shallow) of it. Call the copy circle B. Then if you move the circle A, the circle B should not move

because they have two different center object. But if you implement the deep copy incorrectly, if you move circle A, the circle B will move with it.



Empty Constructors

Also, for the empty constructors, each shape's instance variables need to be initialized. In the case of the Shape class, the name is the only instance variable. You may wonder what Shape's name is; it is "Shape". Other objects may have different names like "Circle", "CSE8B_Line", etc depending if the shape is specifically a Circle, or CSE8B_Line, respectively.

For other classes that have points, you can initialize those points to (0, 0). **Basically you can initialize all the instance variables to some form of 0.** However, the names should be initialized to whatever the shape is (i.e. "Circle", "Rectangle", "ARectangle", etc).

Tips and More Details on Some Classes

This portion of the write-up will help you get started with the PSA and understand the process of writing the shapes. You should begin by implementing **Point.java** and then **Shape.java**. Once you have completed these two, you can move on to all the rest of the shapes. We have provided a more detailed description for **ARectangle** and **Square**. You may expand the suggestions in this section and use them to extend to all the rest of the shapes.

ARectangle.java

Open **ARectangle.java**. It is an abstract class that will be inherited from by both the Square and Rectangle classes. The common information to both Square and Rectangle is that they both have a top left corner from which they draw.

public ARectangle()

The default no-arg constructor of the ARectangle class will create a new ARectangle with an upper left corner at 0, 0 and a name of "ARectangle". Rather than rewriting the code for the constructor three different times (there are three different constructors) let's just defer to another constructor. In this constructor, make a call to the **second constructor** (you can call another constructor with the this keyword) with a name of "ARectangle" and and x and y location of 0. That's all we need to do for this method.

public ARectangle(String name, int x, int y)

In this constructor, we're actually going to defer again to another constructor (notice a pattern?). Call the **third constructor** with the name as a first parameter, and a new Point from the given x and y coordinates. Your call

to this() **must be** the first line in the constructor (or else Java will complain), so create the new Point inline instead of saving it to a local variable.

public ARectangle(String name, Point point)

In this constructor, we will finally save the point we received in the parameter. But first, we need to pass our **name** parameter on to something that will take care of it. Fortunately, the **superclass (Shape)** is going to handle the name field (once you write the code that does so). Start by calling the parent's constructor and passing in the name parameter (you will need to use the **super** keyword to accomplish this). Now we can set the **upperLeft** field with the point we received as a parameter. Use **setUpperLeft()** to set the field instead of modifying it directly.

public ARectangle(ARectangle r)

This is the copy constructor for the ARectangle class. Call the **third constructor** again, passing in r's name and upper left corner (use **getUpperLeft()** instead of accessing it directly).

public Point getUpperLeft()

This is the getter for the upper left corner of the ARectangle. Simply return the upperLeft field here.

private void setUpperLeft(Point upperleft)

Just as we can't hand out references to our upper left corner, whoever gave us a reference to this Point may modify it later, which will change the ARectangle without our permission. Make a deep copy of the point by calling the Point's **copy constructor** and set the upperLeft field to that.

public void move(int deltaX, int deltaY)

All we need to do to move our ARectangle is to move the upper left corner of our ARectangle. Defer to the Point's move() method, which will in turn move the entire ARectangle.

public String toString()

This method should create a String in the form:

<name>: Upper Left Corner: **<upper left's toString()>**

Our subclasses will build off of this template.

public boolean equals(Object o)

This method should return true if the Object o is non-null, an instance of the ARectangle class (use the **instanceof** keyword), and has an equivalent upper left corner to the calling object's upper left corner. You will need to **cast** o to an ARectangle in order to access its upper left corner.

Square.java

public Square()

Just like in ARectangle, we're going to defer to the **second constructor**. Call the second constructor with an x, y, and side length of 0.

public Square(int x, int y, int side)

As before, call the **third constructor**, creating a new Point with x and y, and passing the side through.

public Square(Point upperLeft, int side)

This is the constructor in which we will set our fields. First, call the superclass's constructor with the name "Square" and the upper left corner we've been given. The parents will handle those fields. Set the **side** field (not the parameter) using its setter. You do not need to worry about deep copies here because ARectangle already has us covered.

public Square(Square square)

Call the **third constructor**, passing in the square's upper left and side (using their getters - not accessing them directly!). You do not need to worry about deep copies here because ARectangle already has us covered..

public void setSide(int side)

Set the field side to the value of the parameter side. Remember to use the keyword **this** to distinguish between instance and local variables.

public int getSide()

Return the value of side.

public String toString()

Our superclass already has the basic template for toString(). We just need to add the side length to it. This should be in the form:

<superclass's toString()> Sides: **<side>**

You can call the superclass's toString with **super.toString()**.

public boolean equals(Object o)

Our superclass already covers some of the basic criteria for equivalence. We can start with calling our superclass's equals() and then checking **1)** that o is an instance of Square and **2)** that o has the same side length as the calling object.

public void draw(DrawingCanvas canvas)

For this draw method, we will want to defer to the second draw method, which takes a Color and a boolean. However, we need to first come up with a Color to pass in to that method. Generate a random number for the red, blue, and green components of the Color and make a new color from those. Then call the other draw method passing in true for fill.

public void draw(DrawingCanvas canvas, Color c, boolean fill)

This the method in which we will actually draw our shape. First, we need to check our edge cases: if c is null, set it to black. Then, if fill is true, create a new **FilledRect** object from the objectdraw library, and set its color to c. This will create a full rectangle on the canvas. If fill is false, create a new **FramedRect** object and set its color to c. This will create only the outline of the a rectangle on the canvas. The constructors both take the x and y location of the upper left corner, height and width (which will both be the same length), and the canvas. Each one also has a **setColor()** method you can use to set the color.

Testing your Code

Test files are provided in the starter code given:

YOU DO NOT HAVE TO IMPLEMENT ANYTHING IN ANY OF THESE FILES

TestMickey.java (and TestMickey.html)

TestHouseWithDelays.java and TestHouseWithDelaysController.java (and TestHouseWithDelays.html)

TestSquares.java (and TestSquares.html)

TestTriangle.java (and TestTriangle.html)

TestCSE_8BLine.java (and TestCSE_8BLine.html)

TestRectangle.java (and TestRectangle.html)

TestMickey uses class Shape, class Point, and class Circle. This is a good one to start with to test your class Shape, Circle, and Point.

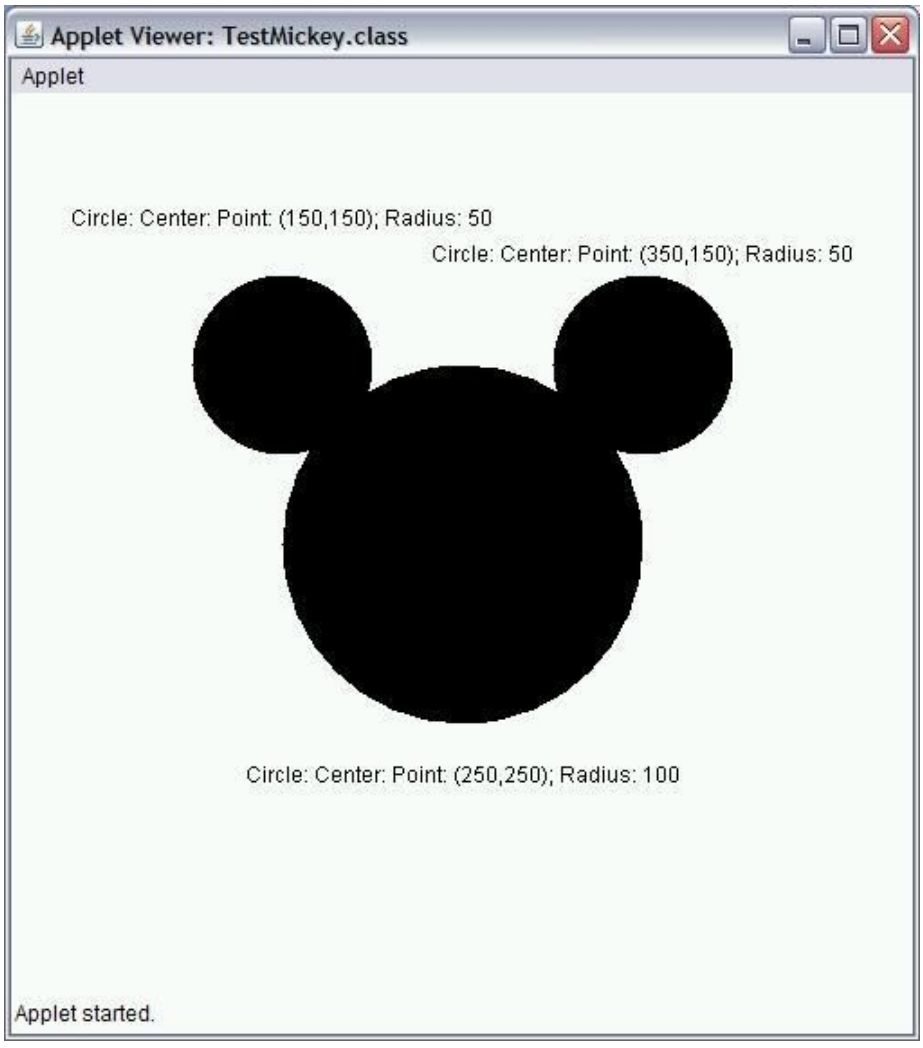
TestSquares, TestTriangle, TestCSE_8BLine, and TestRectangle uses each of the respective shapes so you can check if one of the shapes are being drawn individually.

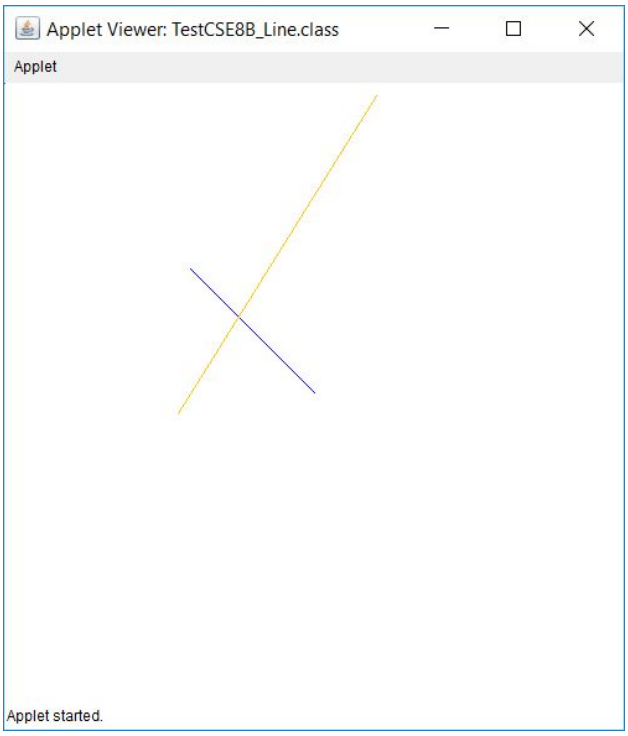
TestHouseWithDelays draws a house using all the various shapes in a delayed fashion so you can see the different objects being drawn. Note: This tester does not use fill to color in the shapes.

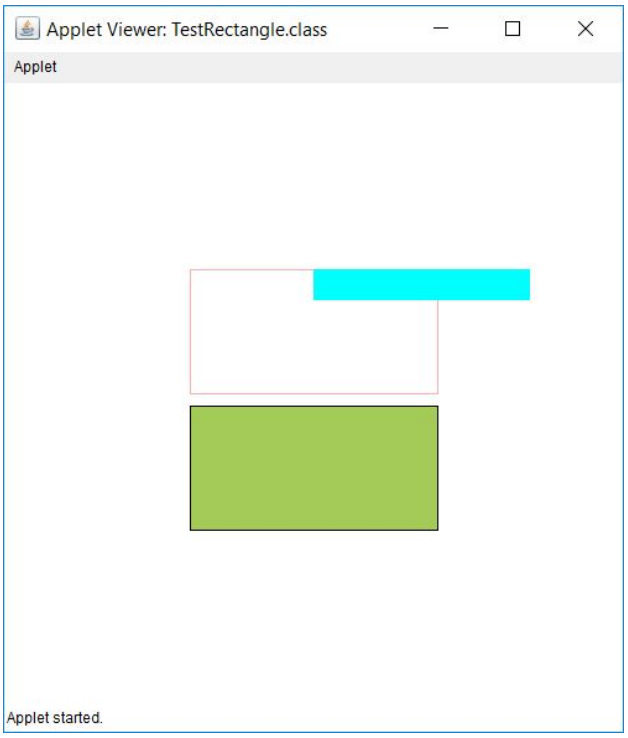
Be sure to take a look inside the test files and try to understand what each tester is trying to accomplish, especially pay attention to the reference type and what constructor is being called to pinpoint errors.

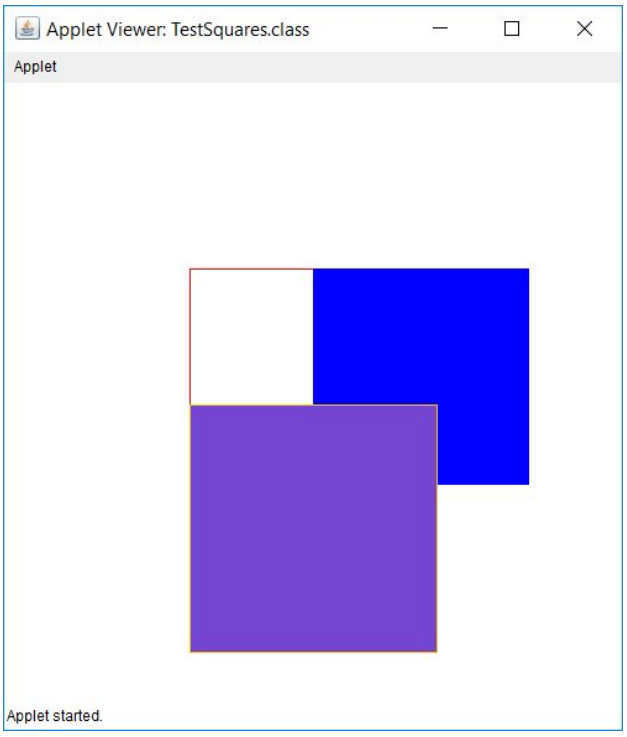
We will compile and use these test programs against your shapes sources to grade this assignment (in addition to using some of our own test cases).

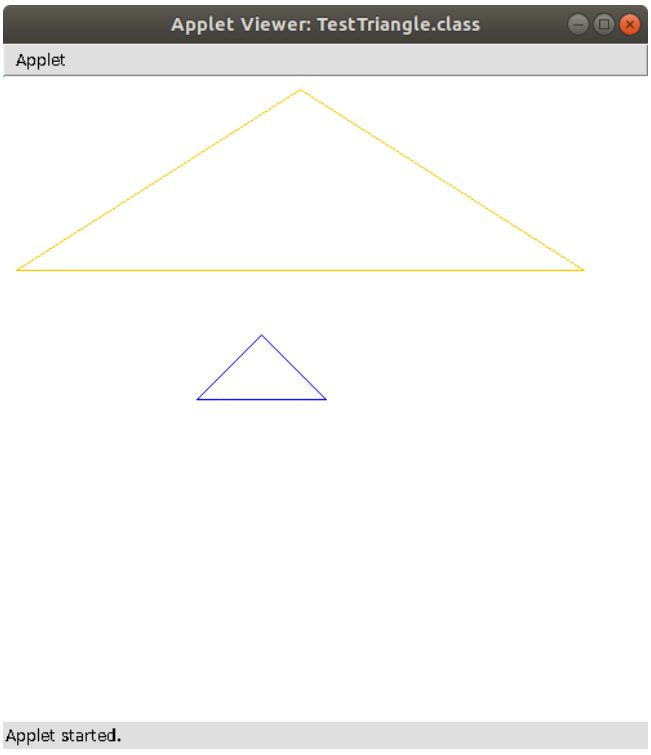
Example screenshots of what each tester looks like on the following pages:

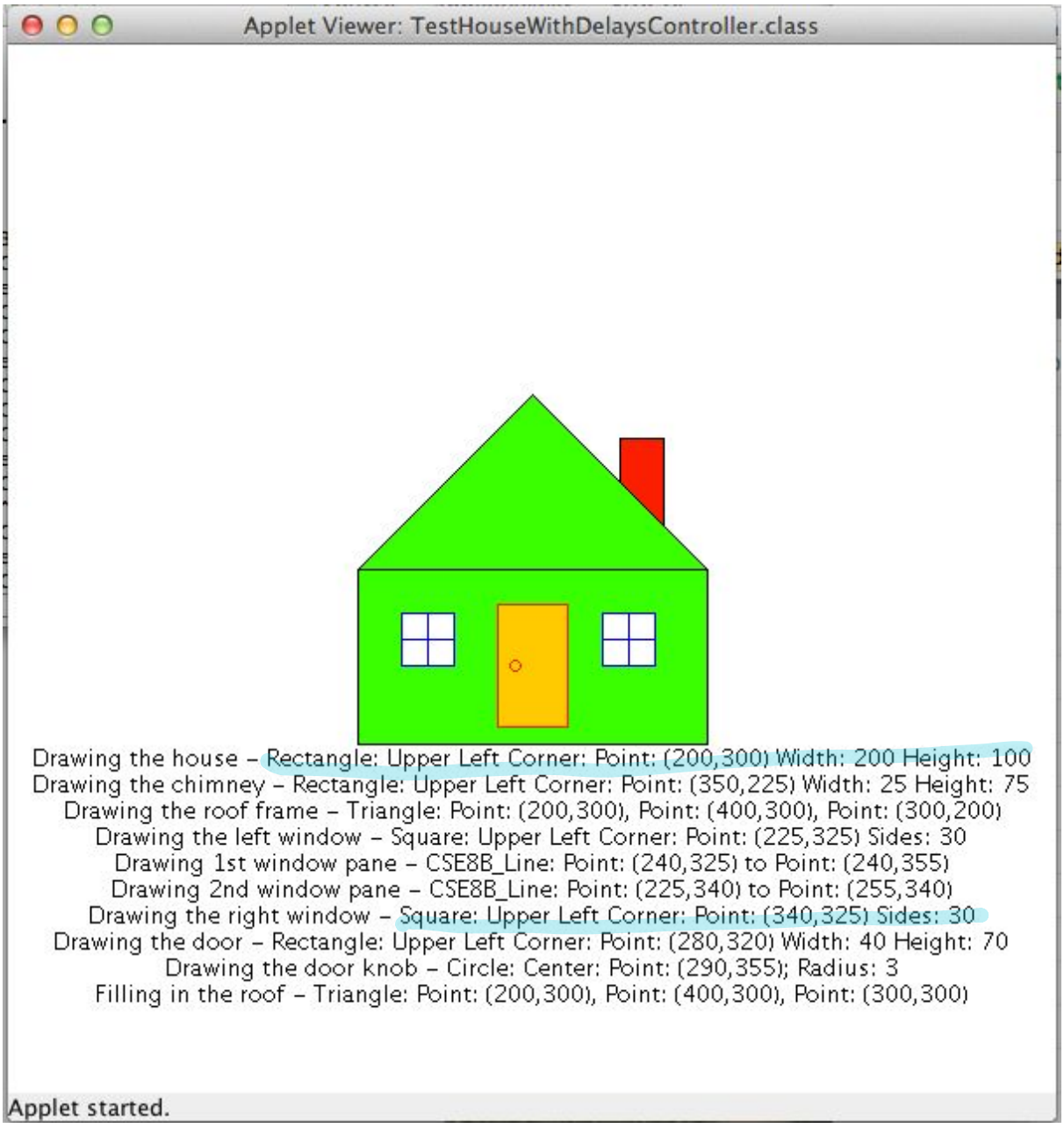



Applet Viewer: TestCSE8B_Line.class
Applet
Applet started.


Applet Viewer: TestRectangle.class
Applet
Applet started.


Applet Viewer: TestSquares.class
Applet
Applet started.


Applet Viewer: TestTriangle.class
Applet
Applet started.

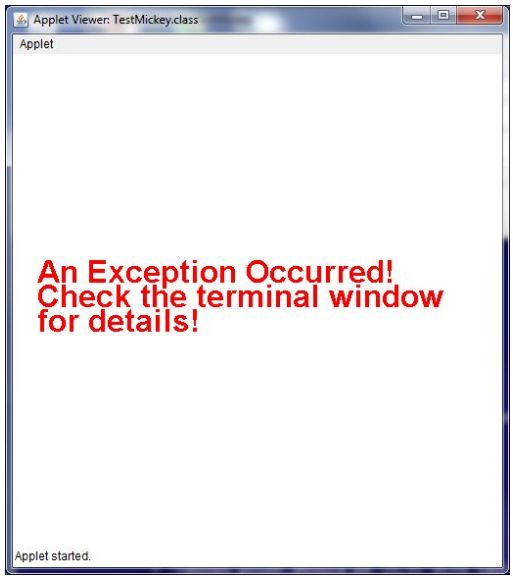


For TestHouseWithDelays, if the text does not fit horizontally you are welcome to change the dimensions of the applet (Keep in mind that this will change your x and y coordinate values so you won't be able to check their correctness against the pictures above). If you have problems with this test, you are allowed to comment out parts of TestHouseWithDelays.java to draw only certain sections of the house at a time to make debugging easier. But in the end all of this test case should work properly.

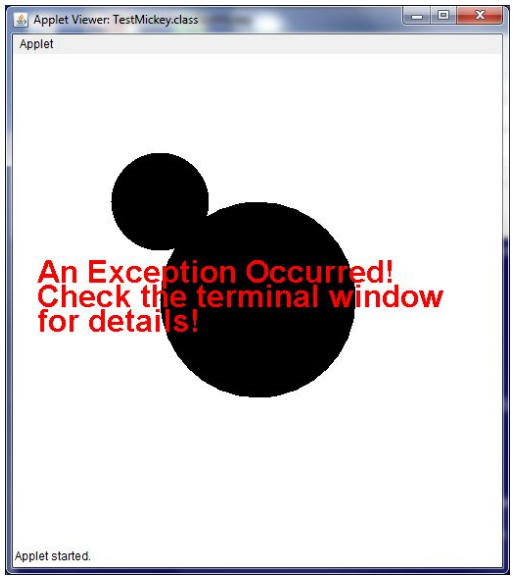
Note: The last Point coordinate represents the value of the last Point to draw a filled Triangle (roof) with multiple (framed) Triangles with a changing Y value of the upper Point. You will see this value change as you run the test program as the roof is filled in

If any exceptions are thrown when you run your program, an error message will pop up in your window. The error message indicates that some sort of exception has been thrown in your terminal. **You must fix these exceptions!** To understand and fix what these exception in the terminal are telling you, you can use the same logic that was on the midterm and discussed in class.

Here are two examples of error messages and exceptions you may see:




```
$ appletviewer TestMickey.html
java.lang.NullPointerException
        at TestMickey.makeMickey(TestMickey.java:54)
        at TestMickey.begin(TestMickey.java:17)
        at objectdraw.WindowController.helpinit(WindowController.java:70)
        at objectdraw.Controller.init(Controller.java:82)
        at sun.applet.AppletPanel.run(AppletPanel.java:435)
        at java.lang.Thread.run(Thread.java:722)
```



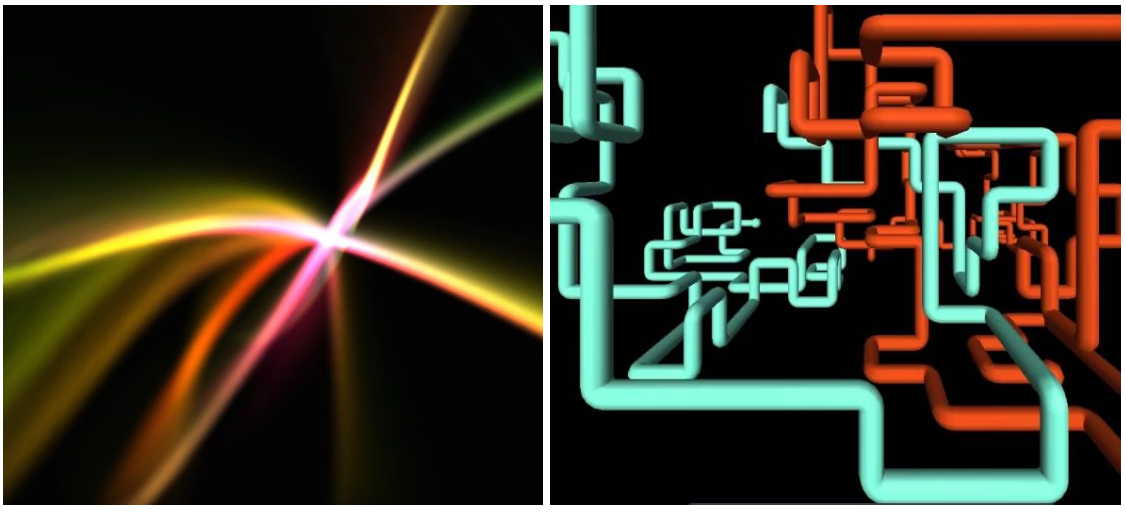

```
$ appletviewer TestMickey.html
java.lang.IllegalStateException:
This should not print!!!
Testing Point equals() with null

        at TestMickey.makeMickey(TestMickey.java:95)
        at TestMickey.begin(TestMickey.java:17)
        at objectdraw.WindowController.helpinit(WindowController.java:70)
        at objectdraw.Controller.init(Controller.java:82)
        at sun.applet.AppletPanel.run(AppletPanel.java:435)
        at java.lang.Thread.run(Thread.java:722)
```

Part 2: Screensaver [10 Points]

Old cathode ray tube (CRT) monitors had a hardware problem. Just like how intense, bright light shined in your eyes may cause you to see that light burned in your eyes (think flash photography), CRT monitors had the screen burn-in problem, or the “ghost image”, where images displayed on the monitor for long periods of time caused the image to stay on the monitor, even when the computer stopped displaying the image.

To solve this issue, major operating systems like Mac OS X and Windows XP implemented a “screensaver”, which is an animated assortment of colors and shapes to make sure no single image displayed too long.



In this section, you will implement a basic screensaver that uses the same implementation tools as you did for Part 1. The file that you will work on is **Screensaver.java**. Look at **`TestHouseWithDelays.java`** for inspiration as you will write a GUI animation that behaves similarly to that program. The idea is to take a given list of Shapes and for each Shape, draw them on a canvas with random colors and positions. In the animation, each shape will be drawn on the canvas one by one.

In the run method, a series of Shapes is being created and placed inside the ArrayList, which is setup to take in Shapes. The window size of the screensaver is set as 800 x 800. (You could see how it is implemented in

the Screensaver.html that we provided you.) The locations and sizes of the shapes are generated randomly in begin method. Notice that the various subclass objects are allowed to be placed inside the ArrayList. Although the reference type is Shape, that is sufficient enough to point to the subclass objects. At the end of the run method, it calls the drawShape method by passing the ArrayList as parameter.

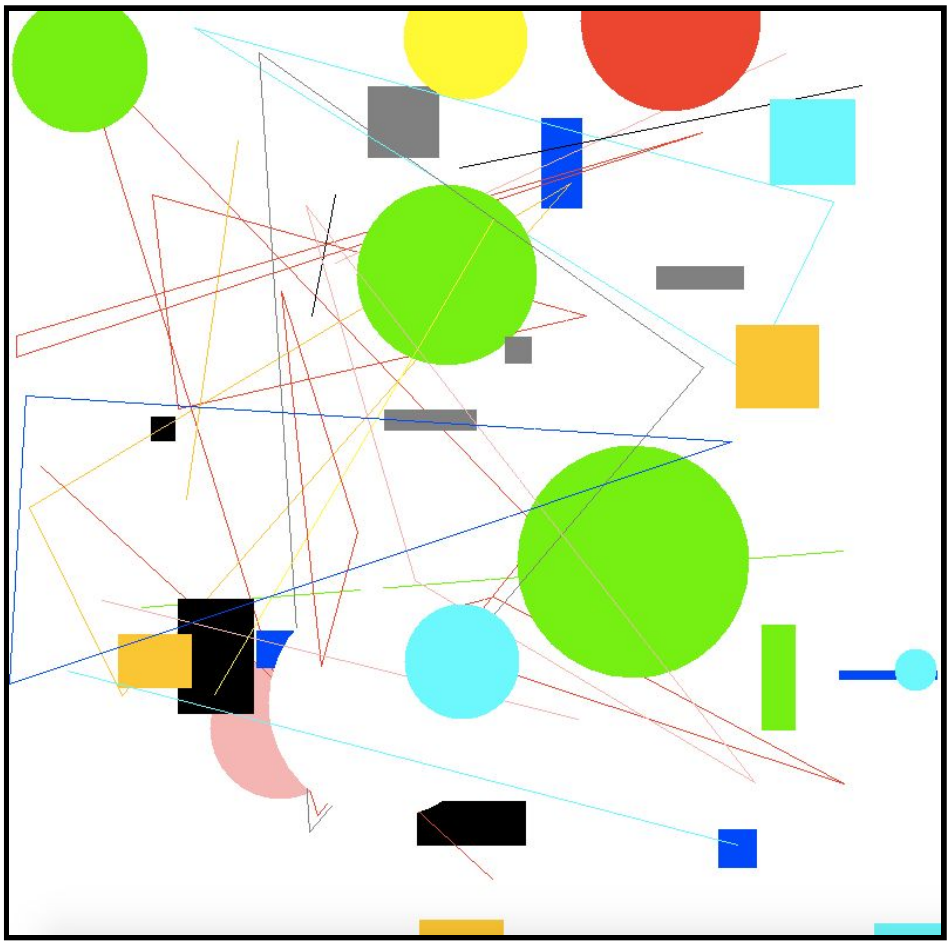
Here is your task. Implement the method drawShape, which as the following signature:

```
public void drawShape( ArrayList<Shape> list)
```

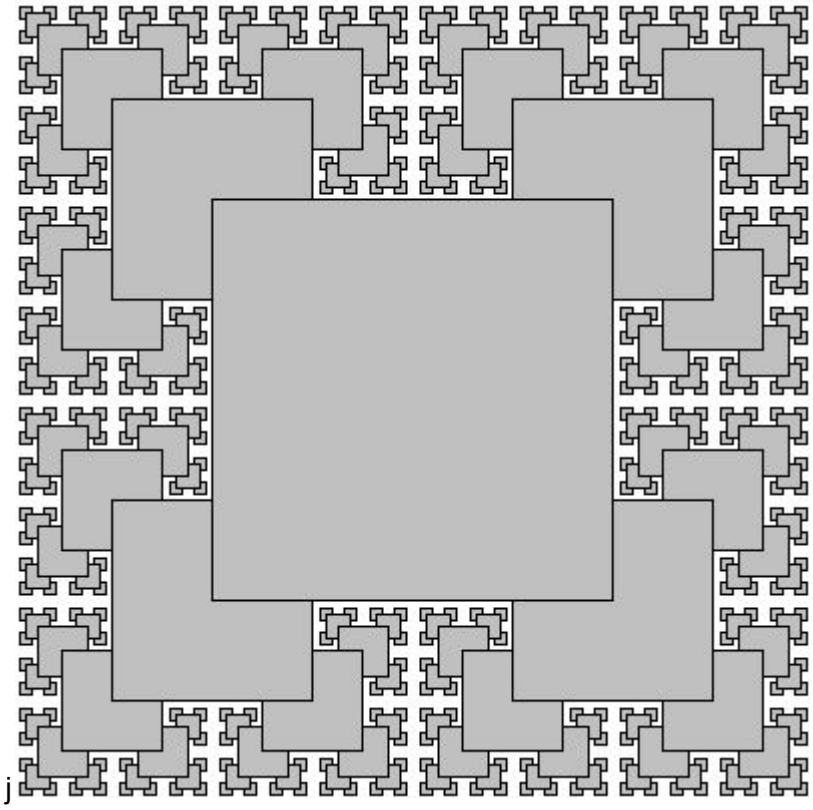
In the drawShape method, use a ***for-each loop*** (***NOT a for-loop, while-loop, or any other mechanism***) to iterate through the parameter ArrayList. Inside the for-each loop, use the Shape reference to call the one-argument draw method, passing just the canvas reference as an argument. Then, call pause(DELAY). In the GUI animation, there should be a shape being drawn, a delay, and then another shape being drawn until the ArrayList runs out of Shapes. Notice that your reference type is Shape, but since all the classes have the same draw method, we are able to implement this with mostly just one for-each loop easily and without error. Hint: It's possible to implement this in less than five lines of code. That is the power of object-oriented programming and polymorphism.

Refer to the beginning of Part 1 for the commands to compile the Java files and run the applet HTML.

An example of screensaver after you implemented and run the code as instructed:



Part 3: Recursive Squares [20 Points]



Recursion is a common technique used by programmers that involves solving a problem by dividing it into smaller problems over and over again, until the problem becomes small enough that it can be reduced to a "trivial" base case. A **recursive method** is one that uses recursion by calling itself within its own method body. You will write a recursive method called `recursiveSquare`, which will draw smaller squares at the four corners of a larger square repeatedly to form a pattern. You will be completing the following method inside of **Recursion.java**:

private void recursiveSquare(Square square, double sizeReduction, int n)

The parameters of this method consist of the following:

- **Square square** -- the square to be drawn
- **double proportion** -- the ratio of the size of the smaller square's sides to the square passed in as a parameter. For example, if `proportion` is 0.50 and a square has sides of size 200, the smaller squares at the four corners created during the next recursive method calls will have sizes of 100.
- **int n** -- an int counter that controls the number of times `recursiveSquare` is called.

The method should:

- Have a base case that returns when some condition is met
- Calculate the new size of the smaller squares to be drawn at the four corners
- Create the new smaller squares accordingly
- Recursively call itself using the new smaller squares
- Draw the square that was passed in as a parameter

Part 4: README.md and Short Response [10 Points]

Write a paragraph on how you tested your Shapes, Screensaver, and Recursion. Examples of your testing procedures might include what features of the Shapes you used to help you run your tests. How do you know your program is correct? How do you know your program is not correct? You may use Java and CSE vocabulary here to discuss your techniques and intuitions about testing.

Write a paragraph about the entire program itself. This program description should NOT be about each class related to the Shapes. That's the file and class headers' job. Rather, the description should talk about what the Shapes do (especially in relation with the Screensaver and the RecursiveSquares). Remember that the audience of the README description is anyone who knows no programming or computer science. This means use absolutely no Java or CSE terms but bird's eye descriptions are fine.

- Use high-level language: describe what the program does, without implementation detail
- Avoid using CS-specific terminologies, so that a person with no CS background can understand
- Describe how to use this program

Short Response

Answer the following questions: (it may be easier to answer these questions after you have completed this assignment)

1. How would you test whether the copy constructors in the shape classes are doing a deep copy instead of a shallow copy?

For example, given:

```
CSE8B_Line line1 = new CSE8B_Line();
CSE8B_Line line2 = new CSE8B_Line(line1);
```

How would you write a test to determine if CSE8B_Line's copy constructor is doing a deep copy?

2. On a similar note, how would you write a test for the equals() method in CSE8B_Line to determine if it is doing a deep comparison vs. a shallow reference comparisons?

3. Please read the tester file TestMickey.java and explain how the Circle class is tested in this tester. Your explanation should focus on the structure of the tester code and which methods from the Circle class is tested.

Style Guidelines (Link) [-10 Points]

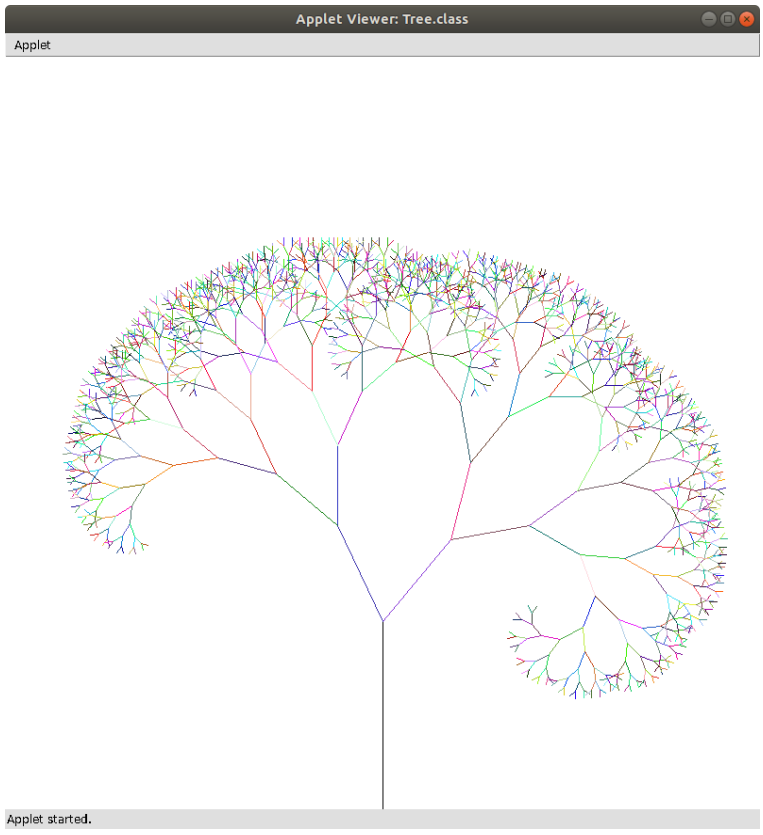
Refer to the website for a complete guideline and set of examples of what your style should look like. In general, having good style is an expectation, rather than something of an accomplishment.

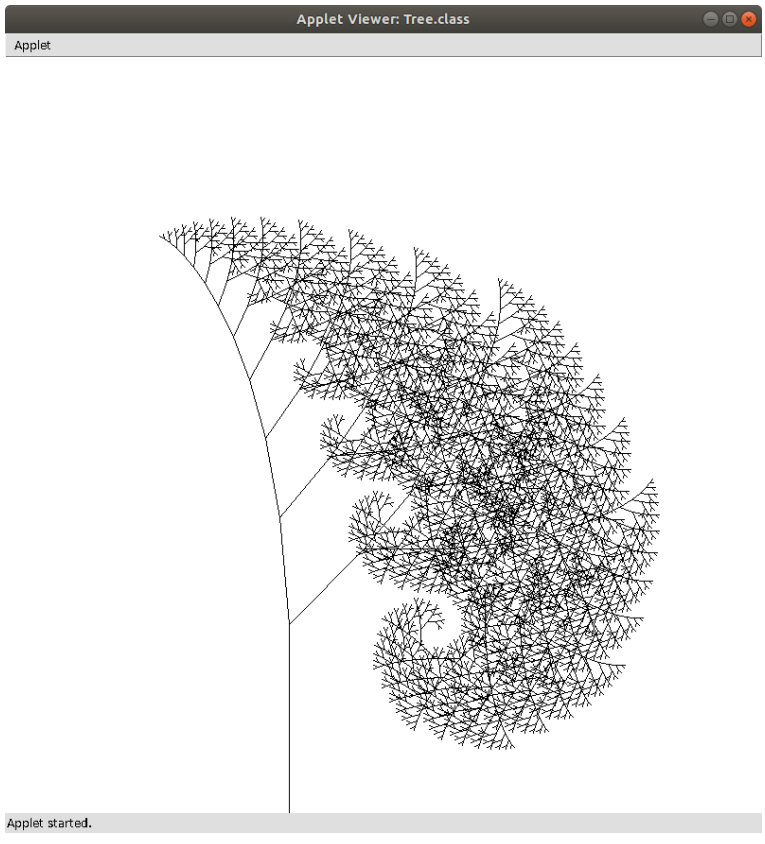
Starting in this assignment, the grading breakdown reflects this expectation. There are ten possible points that can be lost due to bad style, such as but not limited to unindented code and magic numbers. You will also lose significant style points if you do not use **getter and setter methods** to conduct accesses of field variables, rather than directly manipulating the variables.

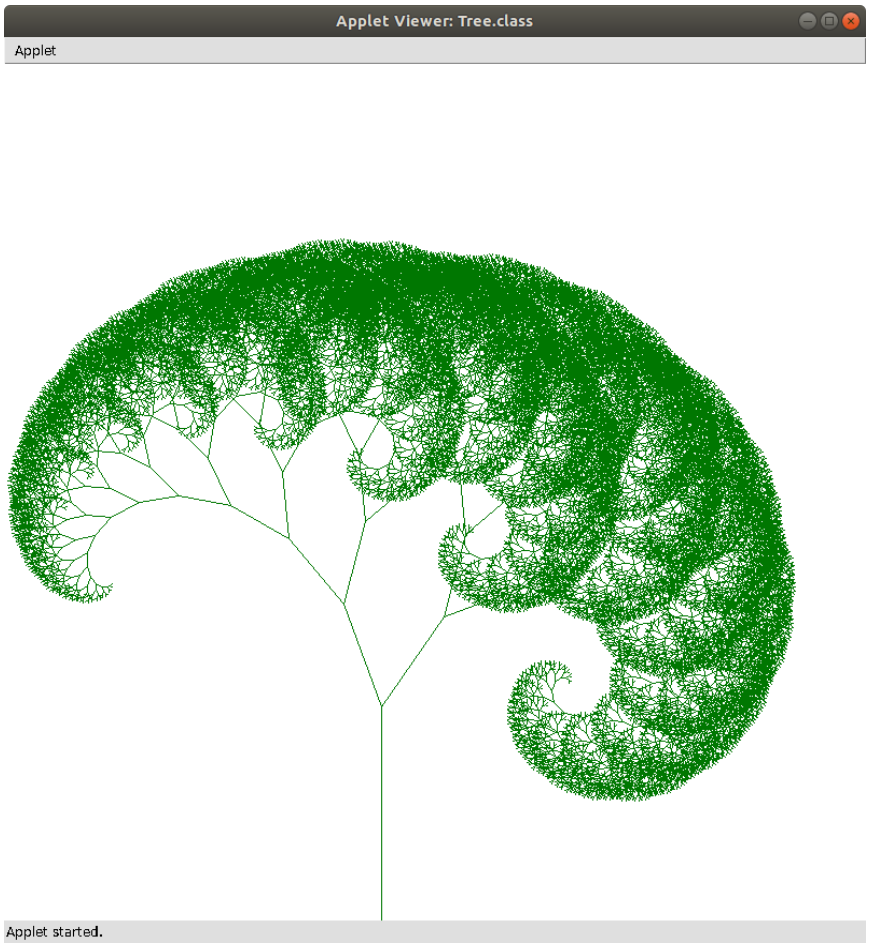
We have scaffolded some **Javadoc method headers** in Point.java. Be sure to take a look at it to reference what Javadoc style looks like as it is now a requirement to get full points.

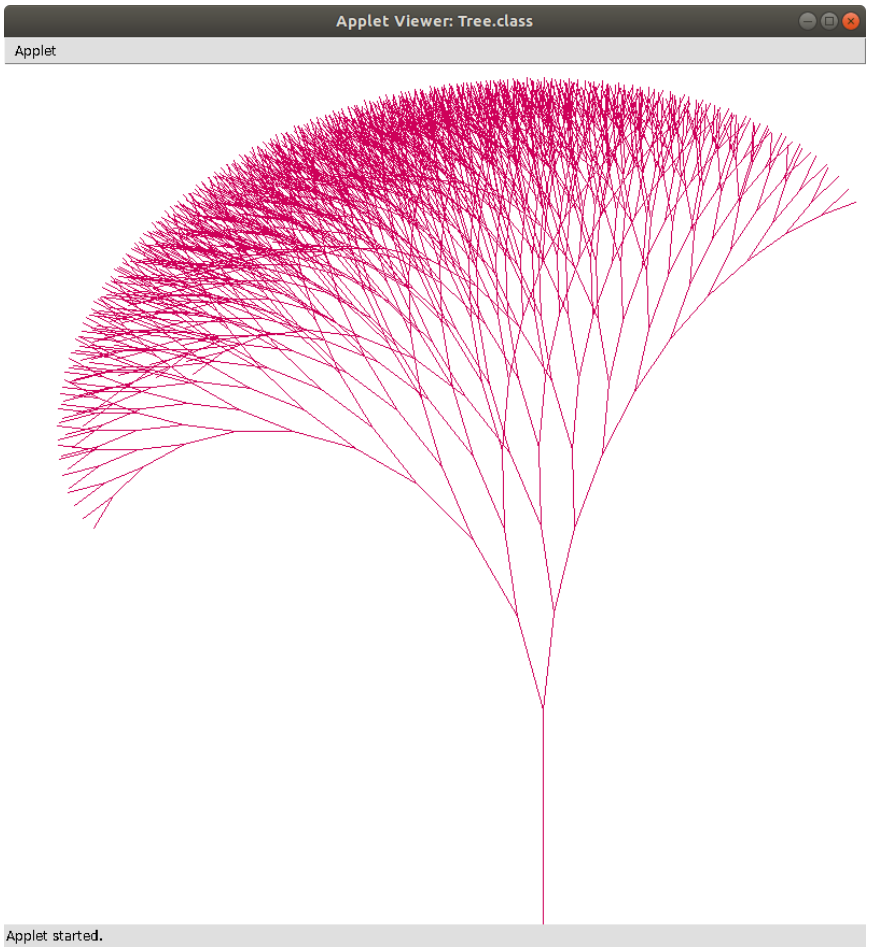
Extra Credit [+10 Points]

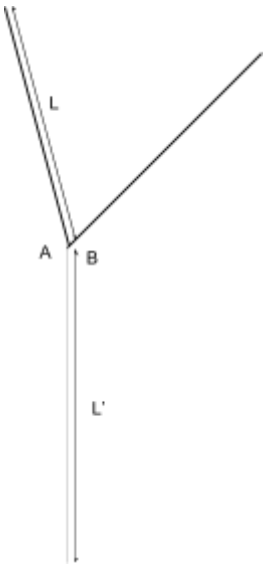
Using recursion, you could draw some very interesting shapes. The following are some example of binary trees (i.e. 2 branches extends out from each branch, but you are free to add more branches) drawn using CSE8B_Line. The extra credit assignment is to implement such a recursive drawing of a tree.





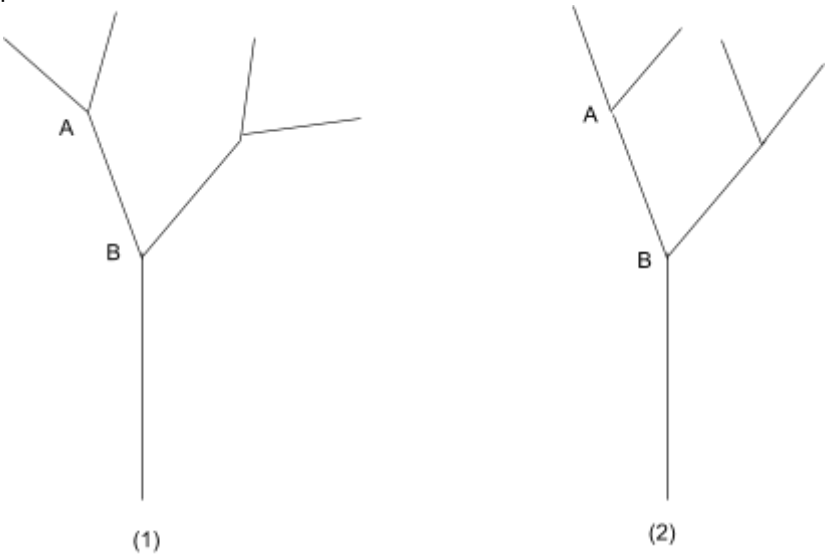

Applet Viewer: Tree.class
Applet
Applet started.


Applet Viewer: Tree.class
Applet
Applet started.



The diagram above shows an example 2 sub-branches drawn by each recursive step (excluding the grey branch). There are many ways to do this, and you're not restricted to any specific implementation. In other words, your tree can be different in color, shape, number of branches, etc. However, your program must follow these specifications:

1. You must write your implementation in Tree.java and Tree.html, and your program must be able to compile with (in the same directory with the rest of your files)
   ```
   > javac -cp ./objectdraw.jar:. Tree.java
   ```
   And your program must show the tree with:
   ```
   > appletviewer Tree.html
   ```
   Hint: Tree.html should be very similar to TestMickey.html, and Tree.java should be similar to TestMickey.java
2. You must use recursion, and there must only be one such recursive method in Tree.java.
3. Recursion must have a base case (i.e. no infinite recursion),
4. At least 2 branches must extend from every branch in your tree, and every sub-branch must be shorter than its parent-branch. (L' > L)
5. Each sub-branch must have a different orientation than other sub-branches, (i.e. angle A must not equal to angle B). Your final tree must be asymmetric.
6. In each recursive step, the angles between sub-branches and the parent-branch must be consistent. As demonstrated by the following diagrams, notice the angle between the left sub-branch and the parent:



Tree (1) is okay, because angle A equals angle B.
Tree (2) is not okay. Angle A and angle B must equal in every recursive level.

Be creative! Try to change up your hyperparameters (e.g. how much shorter is each sub-branch, what angle is each sub-branch, what color, how many recursive steps and what condition to end the recursion, etc), Be sure to not use magic numbers in your code by defining them as static finals.

You may want to check up on Java's built-in trig functions (i.e. Math.sin(), Math.toRadians, etc.) They can be found here:

https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Submitting the Assignment (Link)

- **SUBMITTED PARTS OF CODE THAT DOES NOT COMPILE WILL RESULT IN ZEROES WITHOUT ANY CHANCE OF PARTIAL CREDIT.**
- **SUBMITTED CODE THAT DOES NOT BEHAVE CORRECTLY DUE TO COMPILER VERSION OR OPERATING SYSTEM DIFFERENCES ARE NOT ELIGIBLE FOR REGRADE REQUESTS.**
- **MAKE SURE YOUR SUBMISSION CONTAINS ALL OF THE FILES WITH THE CORRECT CLASS AND METHOD SIGNATURES AND THAT THE CODE COMPILES AND RUNS ON IENG6 LAB MACHINES BEFORE SUBMITTING.**

Submission Files

- README.md
- Point.java
- Shape.java
- CSE8B_Line.java
- Circle.java
- ARectangle.java
- Square.java
- Rectangle.java
- Triangle.java
- Screensaver.java
- Recursion.java

Also submit these files If you attempt the extra credit

- Tree.html
- Tree.java

Maximum Score Possible: 110/100 Points