# Lab Exercise 13: Recursion
# CS 2334

November 16, 2017

## Introduction

In this lab, you will use recursive logic to create fidget spinners based on a fractal triangle, commonly known as a Sierpinski triangle. Recursion has many practical uses outside of graphics, but fractals provide a great way to visualize the concept.

## Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Read existing code and documentation in order to complete an implementation
2. Define the base and recursive cases of a recursive formulation
3. Correctly call a method recursively
4. Create a timer for animation purposes

## Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

## Preparation

1. Import the existing lab 13 implementation into your eclipse workspace.

   (a) Download the lab 13 implementation:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab13/lab13.zip`

   (b) In Eclipse, select *File/Import*

   (c) Select *General/Existing projects into workspace.* Click *Next*

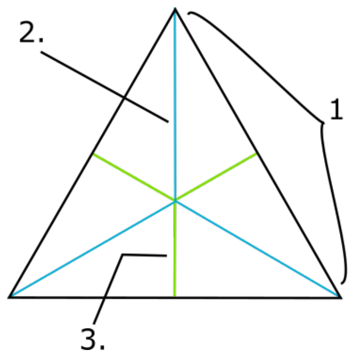
   (d) Select *Select archive file.* Browse to the lab13.zip file. Click *Finish*

# Sierpinski Triangle

The fidget spinner you will create in this lab is based off of the geometry of the Sierpinski Triangle.

> The Sierpinski Triangle is a fractal and attractive fixed set with the overall shape of an equilateral triangle, subdivided recursively into smaller equilateral triangles.
> – from `https://en.wikipedia.org/wiki/Sierpinski_triangle`

For this lab, there are some properties of equilateral triangles that you will need to know:

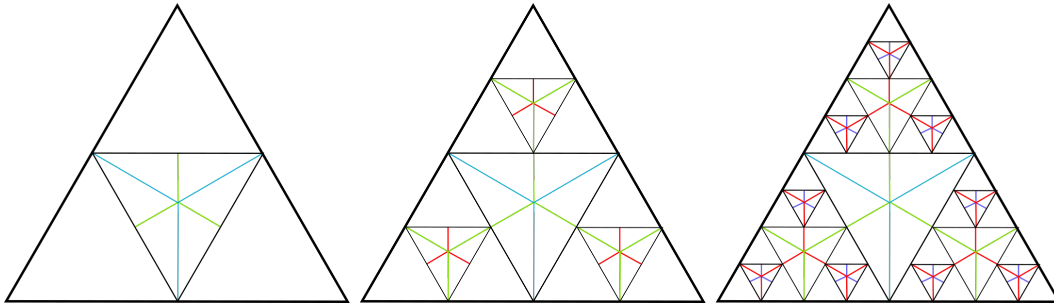



1. A side of the triangle, $s$.
2. The radius of the triangle, $r$.
3. The apothem of the triangle, $a$.

The following relations may be of some use to you:

1. Given the radius, a side is equal to: $s = \sqrt{3} \times r$
2. Equivalently, the radius is equal to: $r = \frac{s}{\sqrt{3}}$
3. The apothem is equal to: $a = \frac{\sqrt{3}}{6} \times s$

The first three iterations of the Sierpinski triangle are shown below (With radii and apothems shown—these won't be drawn in your implementation):



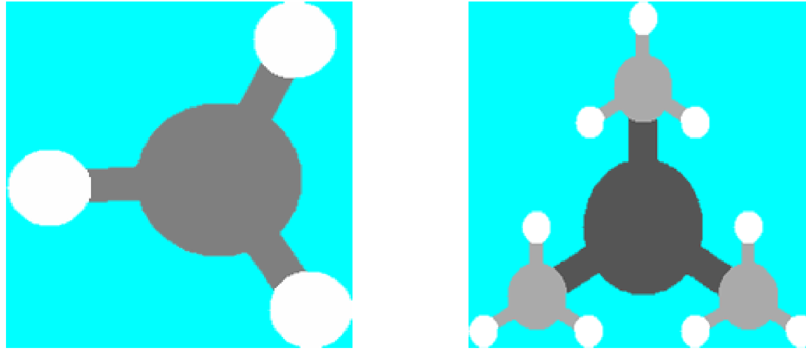
Excluding the black line segments, segments of the same color are the same length. Taking note of this, one can see that the radius of each of the three smaller triangles surrounding a larger triangle is equal to the larger triangle's apothem.

Furthermore, the distance between their center points and the larger triangle's center point is twice the apothem of the larger triangle.

The Sierpinski triangle can be constructed by first starting with an equilateral triangle in the standard orientation (sitting flatly on a side, with one vertex pointing straight up). After drawing this, proceed to recursively draw smaller triangles rotated 180 degrees relative to the orientation of the base triangle. The code you make should be able to draw both Fidget Spinners and a Sierpinski Triangle. To draw the triangle, you can do as follows:

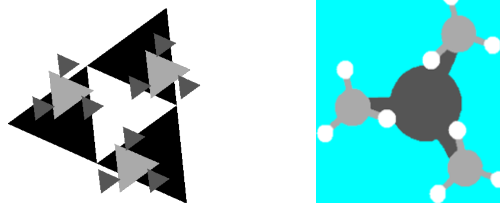
1. The *draw()* method of the *SierpinskiSpinner* class is called.
2. Draw the base triangle (sitting flatly on a side, one vertex pointing straight up).
3. Call the *drawHelper()* method with the correct arguments to draw a triangle in the correct orientation (one vertex pointing straight down), in the correct position, with radius determined using the mathematical relations listed above.
4. In *drawHelper()*, draw the triangle.
5. Compute the apothem of the triangle.
6. Compute the center points of the three surrounding triangles (Remember that all of these triangles are equilateral, so the triangle center points will be evenly spaced 120 degrees around the center triangle).
7. Go to step 3 for the three surrounding triangles

## Sierpinski Spinner

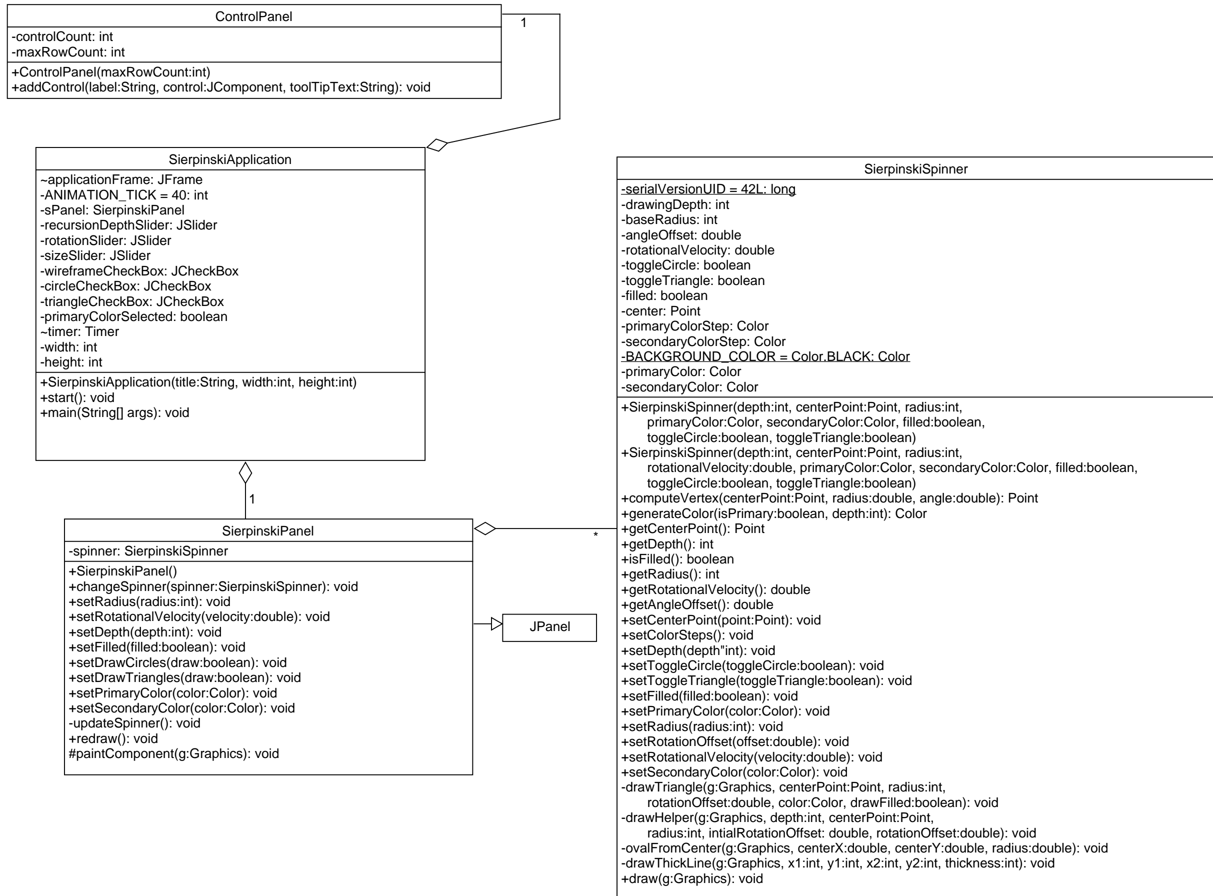


This is what we will actually create for the lab. To create the spinner, we make three main modifications to the original Sierpinski Triangle design:

1. We draw a circle inscribed inside each of the Downward-Facing Triangles. These represent the centers of each fidget spinner.
2. We draw a line between the circles. This represents the arm of each fidget spinner.
3. We add the ability to rotate each sub-triangle around its center triangle. This gives the appearance that each level of spinners is spinning.



# UML



# Lab 13: Specific Instructions

All of the necessary classes are provided in lab13.zip.

1. Look carefully through the existing code and implement any TODOs.
2. Do not add major functionality to the classes beyond what has been specified.
3. Don't forget to document as you go!

## SierpinskiApplication class

- *SierpinskiApplication()* — At the bottom of the constructor a new **javax.swing.Timer** object needs to be created, using *SierpinskiApplication.ANIMATION_TICK* as the delay value. An ActionListener that calls code from a *SierpinskiPanel* instance should cause a step in the animation to occur. This step moves the triangles and redraws the frame.
- To test your program, click on any point in the drawing panel. This will move the Sierpinski Spinner to that point. To change how the spinner is drawn, use the control panel's sliders and toggle boxes. This will allow you to change the size, rotation speed, and recursion depth, as well as allowing to toggle drawing of triangles and circles on/off or drawing them as wireframes.

## SierpinskiSpinner Class

- *draw(Graphics g)* — This method is responsible for actually drawing the Sierpinski triangle to the screen, using the provided Graphics object. Using the Color *SierpinsikiTriangle.BACKGROUND_COLOR*, draw the base triangle. Then, call drawHelper() with the appropriate arguments to draw the first, inner triangle, on top of the base triangle.
- *drawHelper(Graphics g, int depth, Point centerPoint, int radius, double initialRotationOffet, double rotationOffset)* — This is a recursive method that draws a triangle given the provided parameters, and then calls itself to draw surrounding triangles.

  This method takes in the following parameters:

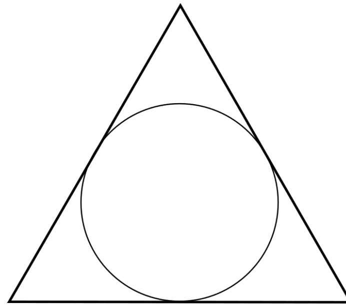
  1. g — The Graphics object to use for drawing.

2. depth — The current recursive depth. The initial call to this method should have depth equal to the triangle's depth field.
3. centerPoint — The center point of the triangle to be drawn.
4. radius — The radius of the triangle to be drawn.
5. initialRotationOffset — The initial rotation of the base triangle draw recursively. This is used to ensure that all triangles are drawn with the same orientation.
6. rotationOffset — This offset is used to calculate the center points of each next level of triangles. By multiplying this offset by a factor of four at each recursive call, we can create a greater rotational offset of the next level of triangles' center points. This creates an effect of spinning on each level of triangles.

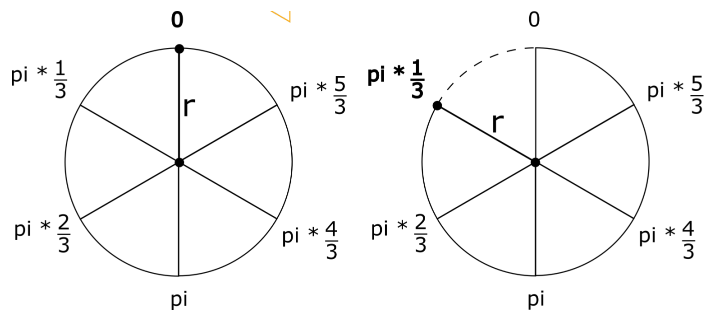
For each call to this method, use the color returned by *generateColor()* to draw the triangle (Triangles are drawn with the primary color).

If the *toggleCircle* boolean has been set to true, you will need to draw an inscribed circle on top of this current triangle; use the color returned by the *generateColor()* method to do so (The circles are drawn with the secondary color).

An inscribed circle looks like the following:

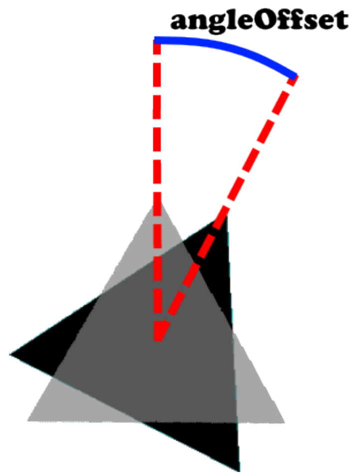


The *SierpinskiSpinner* class has a helper method called *computeVertex()* that takes in a center point, a radius, and an angle. The method starts by generating a point *radius* units directly above the specified center point. The point is then rotated *angle* radians about the center point.



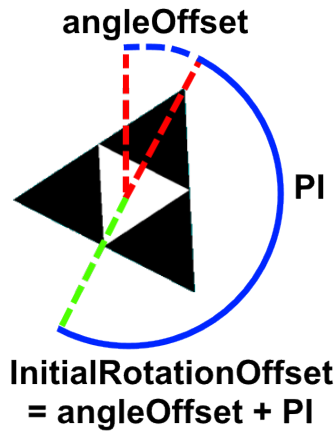

## Elaboration on the Geometry of Recursion

$$Depth = 0$$



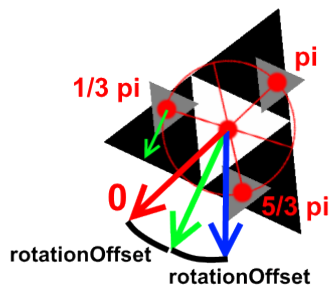

- The base triangle is drawn. The rotation is determined by the angleOffset variable in the SierpinskiSpinner class.
- The gray triangle overlay represents an unrotated triangle, while the black triangle represents a triangle rotated by an angle equal to angleOffset.
- At depth 0, no circles are drawn.

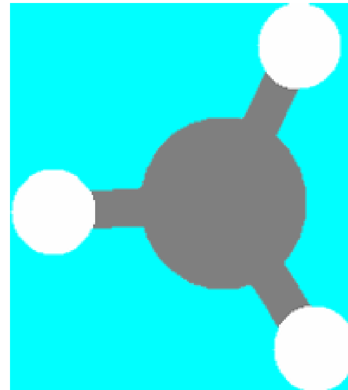
$$Depth = 1$$



- The first triangle (white triangle) created through the drawHelper method is drawn (left panel).
- The angleOffset + PI becomes the initial rotation offset. This determines the orientation of the triangles that we draw.
- At depth 1, a single circle is drawn for the fidget spinner (right panel).

$$Depth = 2$$

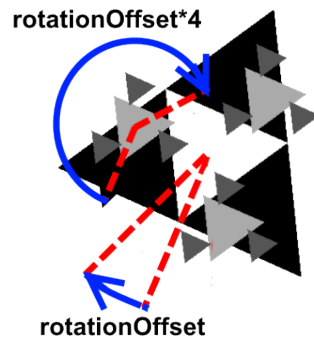


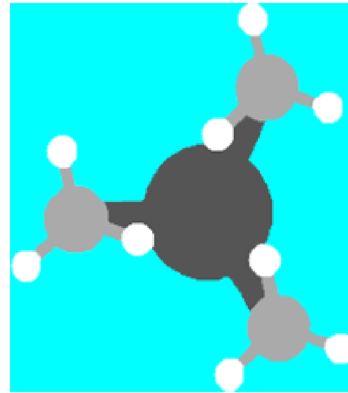


- The new triangles (gray triangles) are drawn around the triangle created from Depth = 1 (white triangle)
- The green arrow is the direction of the white triangle, the same as the gray triangles. The blue arrow is the downward direction, and offset of PI. The red arrow is the direction of the 0 in the unit circle. We use the unit circle to compute the new centerpoints.

- At depth 2, a circle is draw in the center with 3 orbiting circles connected by lines. This is a basic fidget spinner (right panel).

*Depth = 3*



- To compute the centerpoints of the smallest triangles, we follow the same procedure as depth 2. The only difference is that when going between levels of depth we multiply the original rotationOffset by 4.
- At depth 3, there is a fidget spinner at the base with 3 new fidget spinners on top. Because the rotation of the smaller spinners changes faster, we get a spinning effect on all levels.

### SierpinskiPanel Class

- *updateSpinner()* — This method is responsible for computing the new orientation of the spinner. The spinner's angle offset is updated at each animation step. It is equal ot the current offset plus the rotational velocity.

## Hints

The center point for the $i^{th}$ child can be computed from the current triangle's *centerPoint* as follows:

```
computeVertex(centerPoint,
              (int)(apothem * 2),
              initialRotationOffset
                 + (-Math.PI / 3 + ((2 * Math.PI / 3) * i))
                 + rotationOffset);
```

## Final Steps

1. Generate Javadoc using Eclipse.

    - Select *Project/Generate Javadoc...*
    - Make sure that your project (and all classes within it) is selected
    - Select *Private* visibility
    - Use the default destination folder
    - Click *Finish*.

2. Open the *lab13/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that that all of your classes are listed and that all of your documented methods have the necessary documentation.

3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

## Submission Instructions

Before submission, finish testing your program by hand-testing the graphical output. If you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due at 7pm on Saturday, November 18. **Submission must be done through the Web-Cat server.**

- Use the same submission process as you used in lab 1. You must submit your implementation to the *Lab 13: Recursion* area on the Web-Cat server.

## Rubric

The project will be graded out of 100 points. The distribution is as follows:

**Correctness/Testing: 40 points**

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

**Style/Coding: 25 points**

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

**Design/Readability: 35 points**

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

If you do not submit compiled Javadoc for your project, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

**Bonus: up to 5 points**

You will earn one bonus point for every two hours that your assignment is submitted early.

**Penalties: up to 100 points**

You will lose ten points for every minute that your assignment is submitted late. For a submission to be considered *on time*, it must arrive at the server by the designated minute (and zero seconds). For a deadline of 9:00, a submission that arrives at 9:00:01 is considered late (in this context, it is one minute late).

After 15 submissions to Web-Cat, you will be penalized one point for every additional submission.

For labs, the server will continue to accept submissions for three days after the deadline. In these cases, you will still have the benefit of the automatic feedback. However, beyond ten minutes late, you will receive a score of zero.

The grader will make their best effort to select the submission that yields the highest score.