Goals

- To write a library of static methods that performs geometric transforms on polygons.
- To write a program that plots a Sierpinski triangle.
- To design and develop a program that plots a recursive pattern of your own design.

Background

Read Section 2.3 of the textbook. You may also find it instructive to work through some of the other exercises and look at the solutions on the booksite afterwards. You should also familiarize yourself with the `StdDraw` API.

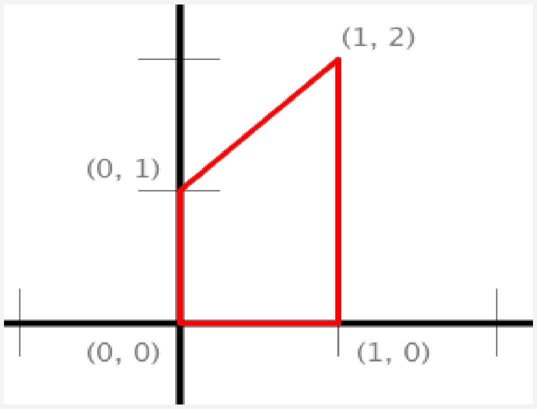
Getting Started

This is an **individual** assignment. Download and unzip the project zip sierpinski.zip, which contains the files you will need for this assignment.

Part I - Geometric Transformation Library

You will write a library of static methods that performs various *geometric transforms* on *polygons*. Mathematically, a polygon is defined by its sequence of vertices (x_0, y_0), (x_1, y_1), (x_2, y_2), …. In Java, we will represent a polygon by storing the *x*- and *y*-coordinates of the vertices in two parallel arrays `x[]` and `y[]`. For example:

```
// a polygon with these four vertices:
// (0, 0), (1, 0), (1, 2), (0, 1)
double x[] = { 0, 1, 1, 0 };
double y[] = { 0, 0, 2, 1 };

// Draw the polygon
StdDraw.polygon(x, y);
```



`Transform2D.java`

Write a two-dimensional transformation library `Transform2D.java` by implementing the following API:

```
public class Transform2D {

    // Returns a new array object that is an exact copy of the given array.
    // The given array is not mutated.
    public static double[] copy(double[] array)

    // Scales the polygon by the factor alpha.
    public static void scale(double[] x, double[] y, double alpha)

    // Translates the polygon by (dx, dy).
    public static void translate(double[] x, double[] y, double dx, double dy)

    // Rotates the polygon theta degrees counterclockwise, about the origin.
    public static void rotate(double[] x, double[] y, double theta)

    // Tests each of the API methods by directly calling them.
    public static void main(String[] args)

}
```

Requirements

- The API expects the angles to be in degrees, but Java's trigonometric functions take the arguments in radians. Use `Math.toRadians()` to convert from degrees to radians.

- The transformation methods `scale()`, `translate()`, and `rotate()` *mutate* the arrays, while `copy()` returns a new array.
- The `main` method **must** test each method of the `Transform2D` library. In other words, you must call each `Transform2D` method from `main`. You should experiment with various data so you are confident that your methods are implemented correctly.
- You can assume the following about the inputs: the arrays passed to `scale()`, `translate()`, and `rotate()` are not `null`, are the same length, and do not contain the values `NaN`, `Double.POSITIVE_INFINITY`, or `Double.NEGATIVE_INFINITY`.
- The array passed to `copy()` is not `null`.
- The values for the parameters `alpha`, `theta`, `dx`, and `dy` are not `NaN`, `Double.POSITIVE_INFINITY`, or `Double.NEGATIVE_INFINITY`.

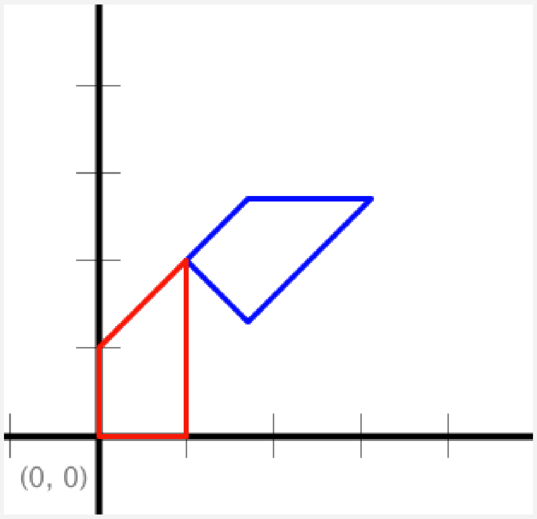
copy()

Copies the given array into a new array object. The given array is not mutated.

The transformation methods (below) mutate a given polygon. This means that the parallel arrays representing the polygon are altered by the transformation methods. It is often useful to save a copy of the polygon before applying a transform.

For example:

```
public static void main(String[] args) {
    // Set the x- and y-scale
    StdDraw.setScale(-5.0, 5.0);

    // Create original polygon
    double[] x = { 0, 1, 1, 0 };
    double[] y = { 0, 0, 2, 1 };

    // Copy original polygon
    double[] cx = copy(x);
    double[] cy = copy(y);

    // Rotate and translate the copy
    rotate(cx, cy, -45.0);
    translate(cx, cy, 1.0, 2.0);

    // Draw the copy in blue
    StdDraw.setPenColor(StdDraw.BLUE);
    StdDraw.polygon(cx, cy);

    // Draw the original polygon in red
    StdDraw.setPenColor(StdDraw.RED);
    StdDraw.polygon(x, y);
}
```



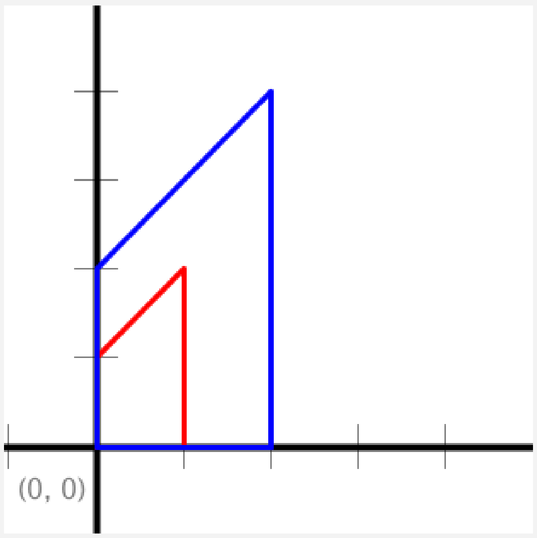
scale()

Scales the coordinates of each vertex (x_i, y_i) by a factor α.

- $x_i' = \alpha x_i$
- $y_i' = \alpha y_i$

An example of testing code for `scale()` is provided below. However, we highly encourage you to experiment with various **values** to confirm that your methods work as required.

```
public static void main(String[] args) {
    // Set the x- and y-scale
    StdDraw.setScale(-5.0, +5.0);

    // Create polygon
    double[] x = { 0, 1, 1, 0 };
    double[] y = { 0, 0, 2, 1 };

    // Draw original polygon in red
    StdDraw.setPenColor(StdDraw.RED);
    StdDraw.polygon(x, y);

    // Scale polygon by 2.0
    scale(x, y, 2.0);

    // Draw scaled polygon in blue
    StdDraw.setPenColor(StdDraw.BLUE);
    StdDraw.polygon(x, y);
}
```



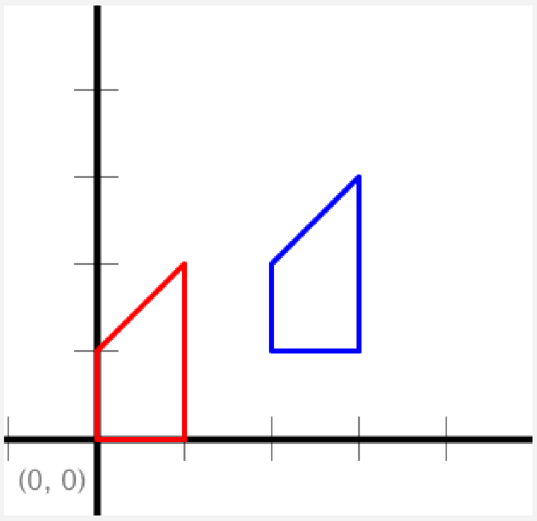
`translate()`

Translates each vertex (x_i, y_i) by a given offset (dx, dy).

- $x_i' = x_i + dx$
- $y_i' = y_i + dy$

An example of testing code for `translate()` is provided below. However, we highly encourage you to experiment with various **values** to confirm that your methods work as required.

```
public static void main(String[] args) {
    // Set the x- and y-scale
    StdDraw.setScale(-5.0, +5.0);

    // Create polygon
    double[] x = { 0, 1, 1, 0 };
    double[] y = { 0, 0, 2, 1 };

    // Draw original polygon in red
    StdDraw.setPenColor(StdDraw.RED);
    StdDraw.polygon(x, y);

    // Translate polygon by
    // 2.0 in the x-direction
    // 1.0 in the y-direction
    translate(x, y, 2.0, 1.0);

    // Draw translated polygon in blue
    StdDraw.setPenColor(StdDraw.BLUE);
    StdDraw.polygon(x, y);
}
```



`rotate()`

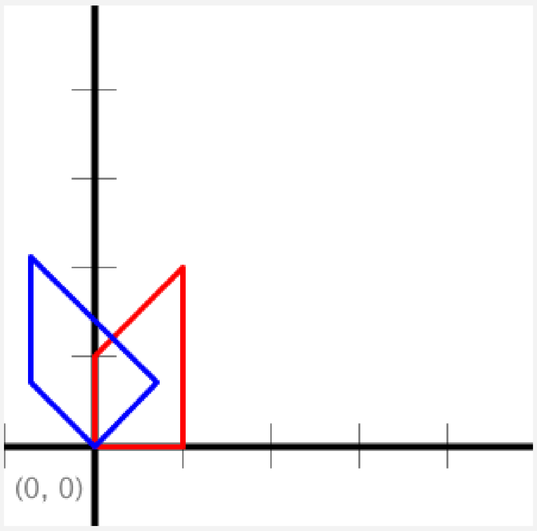
Rotates each vertex (x_i, y_i) by *θ* degrees counterclockwise around the origin.

- $x_i' = x_i \cos\theta - y_i \sin\theta$
- $y_i' = y_i \cos\theta + x_i \sin\theta$

Note in the equations, x_i' and y_i' depend on the x_i and y_i, respectively. In your implementation, you may want to make a **copy** of the *x* and *y* arrays before you compute the *x'* and *y'* arrays!

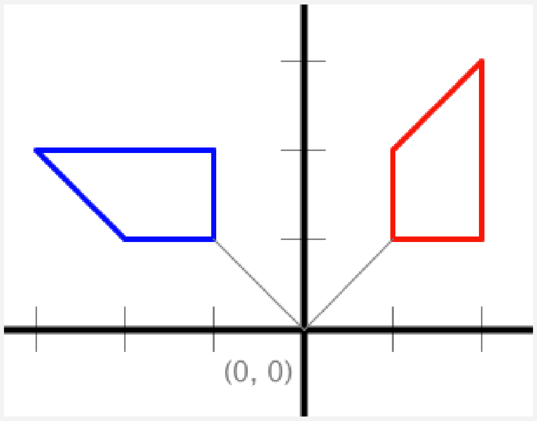
An example of testing code for `rotate()` is provided below. However, we highly encourage you to experiment with various **values** to confirm that your methods work as required.

```
public static void main(String[] args) {
    // Set the x- and y-scale
    StdDraw.setScale(-5.0, +5.0);

    // Create polygon
    double[] x = { 0, 1, 1, 0 };
    double[] y = { 0, 0, 2, 1 };

    // Draw original polygon in red
    StdDraw.setPenColor(StdDraw.RED);
    StdDraw.polygon(x, y);

    // Rotate polygon by 45 degrees ccw
    rotate(x, y, 45.0);

    // Draw rotated polygon in blue
    StdDraw.setPenColor(StdDraw.BLUE);
    StdDraw.polygon(x, y);
}
```



A polygon does not have to be located at the origin in order to rotate it; you can rotate any polygon about the origin using the same method. For example:

```
public static void main(String[] args) {
    // Set the x- and y-scale
    StdDraw.setScale(-5.0, +5.0);

    // Create polygon
    double[] x = { 1, 2, 2, 1 };
    double[] y = { 1, 1, 3, 2 };

    // Draw original polygon in red
    StdDraw.setPenColor(StdDraw.RED);
    StdDraw.polygon(x, y);
```



```
    // Rotate polygon by 90 degrees ccw
    rotate(x, y, 90.0);

    // Draw rotated polygon in blue
    StdDraw.setPenColor(StdDraw.BLUE);
    StdDraw.polygon(x, y);
}
```

Note: All drawings were generated with the coordinate axes, but you will not see them using only the code provided. You do not need to plot the axes.

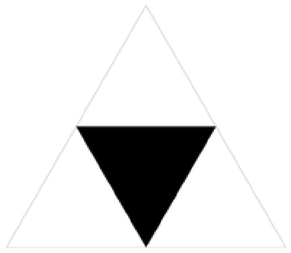
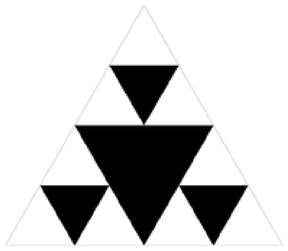
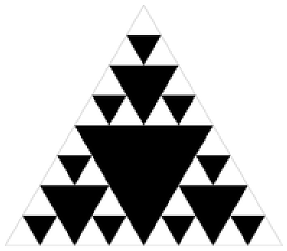
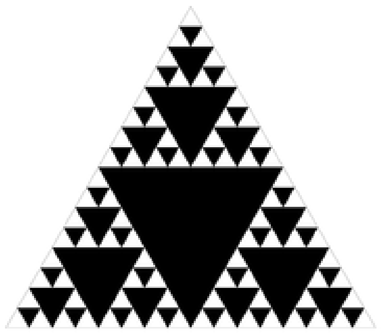
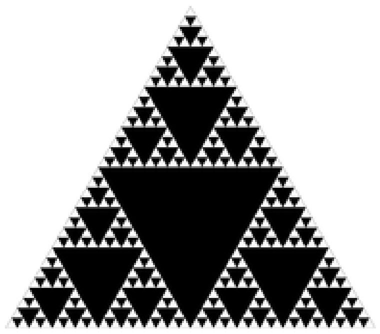
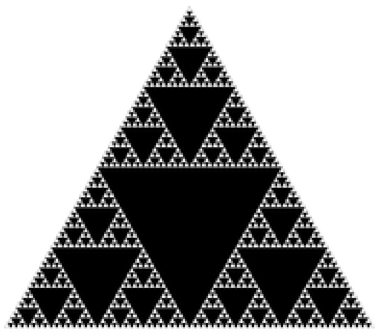
Rotating Around Arbitrary Point
Even though the rotate code above will only rotate polygons about the origin, you could easily transform about any other point (*px*, *py*) using a simple technique. First, translate the polygon by (-*px*, -*py*) so that the rotation point is now at the origin. Next, use your `rotate()` function above to rotate about the origin. Finally, move the polygon back to the rotation point by simply translating by (*px*, *py*). Bam – three lines of code, and you're done! This might be helpful if you draw polygons in your art project below.

`main()`

Tests **each** method of the `Transform2D` library by calling it. You should experiment with various data so you are confident that your methods are implemented correctly. Feel free to draw to standard draw using different polygons. You **should not** expect any command-line arguments.

Part II - Sierpinski Triangle

The Sierpinski triangle[1] is an example of a fractal pattern like the H-tree pattern from Section 2.3 of the textbook.

 Order 1	 Order 2	 Order 3
 Order 4	 Order 5	 Order 6

The Polish mathematician Wacław Sierpiński described the pattern in 1915, but it has appeared in Italian art since the 13th century. Though the Sierpinski triangle looks complex, it can be generated with a short recursive function.

Your main task is to write a recursive function `sierpinski()` that plots a Sierpinski triangle of order n to standard drawing. Think recursively: `sierpinski()` should draw one filled equilateral triangle (pointed downwards) and then call itself recursively three times (with an appropriate stopping condition). It should draw 1 filled triangle for n = 1; 4 filled triangles for n = 2; and 13 filled triangles for n = 3; and so forth.

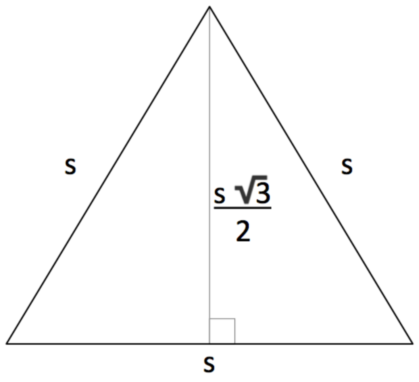
`Sierpinski.java`

When writing your program, exercise modular design by organizing it into four functions, as specified in the following API:

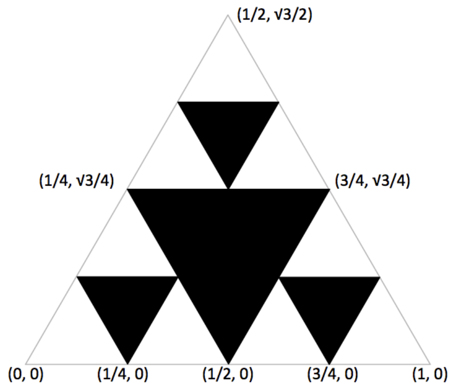
[1] The Polish mathematician Wacław Sierpiński described the pattern in 1915, but it has appeared in Italian art since the 13th century. Though the Sierpinski triangle looks complex, it can be generated with a short recursive function.

```
public class Sierpinski {

    // Height of an equilateral triangle with the specified side length.
    public static double height(double length)

    // Draws a filled equilateral triangle with the specified side length
    // whose bottom vertex is (x, y).
    public static void filledTriangle(double x, double y, double length)

    // Draws a Sierpinski triangle of order n, such that the largest filled
    // triangle has the specified side length and bottom vertex (x, y).
    public static void sierpinski(int n, double x, double y, double length)

    // Takes an integer command-line argument n;
    // draws the outline of an upwards equilateral triangle of length 1
    // whose bottom-left vertex is (0, 0) and bottom-right vertex is (1, 0);
    // and draws a Sierpinski triangle of order n that fits inside the outline.
    public static void main(String[] args)

}
```

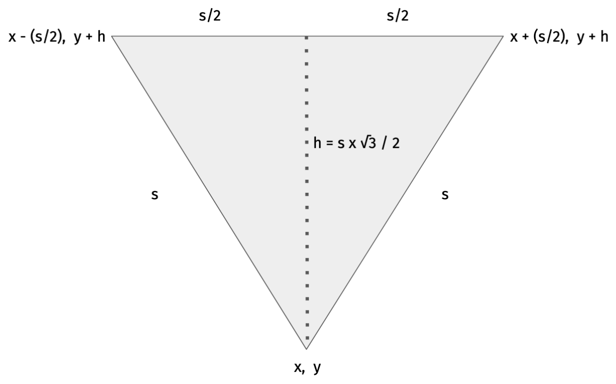
The formula for the *height* of an equilateral triangle of side length s is $h = s \times \frac{\sqrt{3}}{2}$.



Here is the layout of the initial equilateral triangle. The top vertex lies at $\left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$.



Here is the layout of an inverted equilateral triangle.



Requirements

1. To draw a *filled* equilateral triangle, you should call the method `StdDraw.filledPolygon()` with appropriate arguments.
2. To draw an *unfilled* equilateral triangle, you should call the method `StdDraw.polygon()` with appropriate arguments.
3. You **must not** call `StdDraw.save()`, `StdDraw.setCanvasSize()`, `StdDraw.setXscale()`, `StdDraw.setYscale()`, or `StdDraw.setScale()`. These method calls interfere with grading.
4. You may use any colors that you like to draw either the outline triangle or the filled triangles, provided it contrasts with the white background.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps. Note that your final `Sierpinski.java` program should not be very long (no longer than `Htree.java`, not including comments and blank lines).

- Review Htree.java from the textbook and lecture.

- Write a (non-recursive) function `height()` that takes the length of the side of an equilateral triangle as an argument and returns its height. The body of this method should be a one-liner.
    - **Test** your `height()` function. This means try your `height()` function with various values. Does it return the correct calculation?

- In `main()`, draw the outline of the initial equilateral triangle. Use the `height()` function to calculate the vertices of the triangle.

- Write a (nonrecursive) function `filledTriangle()` that takes three (3) arguments (x, y, `length`) and draws a filled equilateral triangle (pointed downward) with the specified side length and the bottom vertex at (*x*, *y*).

- ○ To **test** your function, write `main()` so that it calls `filledTriangle()` a few times with different arguments. You will be able to use this function without modification in `Sierpinski.java`.

- Ultimately, you must write a *recursive* function `sierpinski()` that takes four (4) arguments (`n`, `x`, `y`, `length`) and plots a Sierpinski triangle of order `n`, whose largest triangle has the specified side length and bottom vertex (*x*, *y*). However, to implement this function, use an *incremental* approach:
  - Write a recursive function `sierpinski()` that takes one argument `n`, prints the value `n`, and then calls itself three times with the value `n-1`. The recursion should stop when `n` becomes 0.

  - To **test** your function, write `main()` so that it takes an integer command-line argument `n` and calls `sierpinski(n)`. Ignoring whitespace, you should get the following output when you call `sierpinski()` with `n` ranging from 0 to 5. Make sure you understand how this function works, and why it prints the numbers in the order it does.

```
> java-introcs Sierpinski 0    > java-introcs Sierpinski 4
[no output]                    4
                               3 2 1 1 1 2 1 1 1 2 1 1 1
> java-introcs Sierpinski 1    3 2 1 1 1 2 1 1 1 2 1 1 1
1                              3 2 1 1 1 2 1 1 1 2 1 1 1

> java-introcs Sierpinski 2    > java-introcs Sierpinski 5
2                              5
1                              4 3 2 1 1 1 2 1 1 1 2 1 1 1
1                                3 2 1 1 1 2 1 1 1 2 1 1 1
1                                3 2 1 1 1 2 1 1 1 2 1 1 1
                               4 3 2 1 1 1 2 1 1 1 2 1 1 1
> java-introcs Sierpinski 3      3 2 1 1 1 2 1 1 1 2 1 1 1
3                                3 2 1 1 1 2 1 1 1 2 1 1 1
2 1 1 1                        4 3 2 1 1 1 2 1 1 1 2 1 1 1
2 1 1 1                          3 2 1 1 1 2 1 1 1 2 1 1 1
2 1 1 1                          3 2 1 1 1 2 1 1 1 2 1 1 1
```

  - Modify `sierpinski()` so that in addition to printing `n`, it also prints the length of the triangle to be plotted. Your function should now take two arguments: `n` and `length`. The initial call from `main()` should be to `sierpinski(n, 0.5)`, since the largest Sierpinski triangle has side length 0.5. Each successive level of recursion halves the length. Ignoring whitespace, your function should produce the following output:

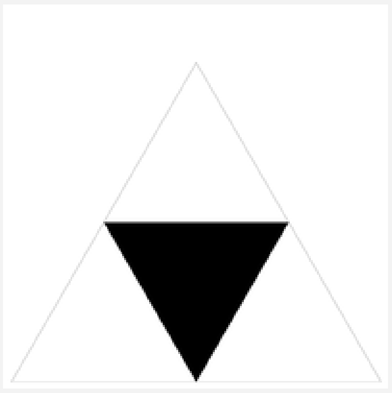
```
> java-introcs Sierpinski 0
[no output]

> java-introcs Sierpinski 1
1 0.5

> java-introcs Sierpinski 2
```

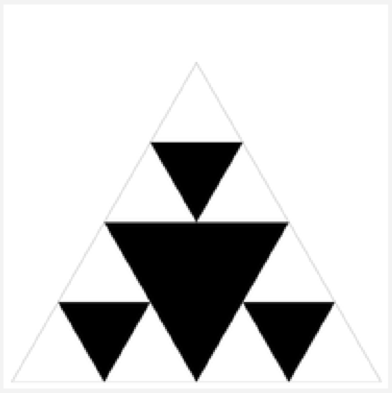
```
2 0.5
1 0.25
1 0.25
1 0.25

> java-introcs Sierpinski 3
3 0.5
2 0.25  1 0.125  1 0.125  1 0.125
2 0.25  1 0.125  1 0.125  1 0.125
2 0.25  1 0.125  1 0.125  1 0.125

> java-introcs Sierpinski 4
4 0.5
3 0.25  2 0.125  1 0.0625  1 0.0625  1 0.0625
        2 0.125  1 0.0625  1 0.0625  1 0.0625
        2 0.125  1 0.0625  1 0.0625  1 0.0625
3 0.25  2 0.125  1 0.0625  1 0.0625  1 0.0625
        2 0.125  1 0.0625  1 0.0625  1 0.0625
        2 0.125  1 0.0625  1 0.0625  1 0.0625
3 0.25  2 0.125  1 0.0625  1 0.0625  1 0.0625
        2 0.125  1 0.0625  1 0.0625  1 0.0625
        2 0.125  1 0.0625  1 0.0625  1 0.0625
```

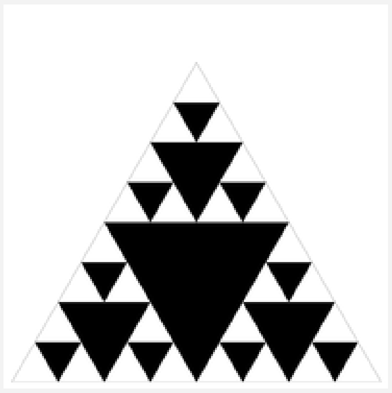
    - Modify `sierpinski()` so that it takes **four** (4) arguments (`n`, `x`, `y`, `length`) and plots a Sierpinski triangle of order `n`, whose largest triangle has the specified side length and bottom vertex (*x*, *y*). Start by drawing Sierpinski triangles with pencil and paper. What values need to change between each recursive call?

- **Remove all print statements before submitting to TigerFile.**

Below are the target Sierpinski triangles for different values of `n`.

```
> java-introcs Sierpinski 1
```


```
> java-introcs Sierpinski 2
```


```
> java-introcs Sierpinski 3
```


Part III - Create Your Own Art

`Art.java`

In this part you will create a program `Art.java` that produces a recursive drawing of your own design. This part is meant to be fun, but here are some guidelines in case you're not so artistic.

A very good approach is to first choose a self-referential pattern as a target output. Check out the graphics exercises in Section 2.3. Here are some of our favorite student submissions from a previous year. See also the Famous Fractals in Fractals Unleashed for some ideas. Here is a list of fractals, by Hausdorff dimension. Some pictures are harder to generate than others (and some require trigonometry).

Requirements

1. `Art.java` must take **one** (1) integer command-line argument `n` that controls the depth of recursion.
2. Your drawing **must** stay within the drawing window when `n` is between 1 and 6 inclusive. (The autograder will not test values of `n` outside of this range.)
3. You may not change the size of the drawing window (but you may change the scale). Do not add sound.
4. Your drawing can be a geometric pattern, a random construction, or anything else that takes advantage of recursive functions.
5. Optionally, you may use the `Transform2D` library you implemented in Part 1. You may also define additional geometric transforms in `Art.java`, such as sheer, reflect across the *x*- or *y*- axis, or rotate about an arbitrary point (as opposed to the origin).
6. Your program must be organized into at least *three* separate functions, including `main()`. All functions except `main()` must be private.
7. For full credit, `Art.java` must not be something that could be easily rewritten to use loops in place of recursion, and some aspects of the recursive function-call tree (or how parameters or overlapping are used) must be distinct from the in-class examples (`HTree`, `NestedCircles`, etc.). You must do **at least two** of the following things to get full credit on `Art.java` (and doing more ***may*** yield a small amount of extra credit):
   a. call one or more `Transform2D` methods
   b. use different parameters than examples: `f(n, x, y, size)`
   c. use different `StdDraw` methods than examples (e.g., ellipses, arcs, text; take a look at the `StdDraw` API)
   d. have non-constant number of recursive calls per level (e.g., conditional recursion)
   e. have mutually recursive methods
   f. have multiple recursive methods
   g. use recursion that doesn't always recur from level `n` to level `n-1`
   h. draw between recursive calls, not just before or after all recursive calls
   i. use recursive level for secondary purpose (e.g., level dictates color)
8. Contrast this with the examples `HTree`, `Sierpinski`, and `NestedCircles`, which have very similar structures to one another.

9. You will also lose points if your artwork can be created just as easily without recursion (such as `Factorial.java`). If the recursive function-call tree for your method is a straight line, it probably falls under this category.
10. You may use GIF, JPG, or PNG files in my artistic creation. If you do, be sure to submit them along with your other files. Make it clear in your `readme.txt` what part of the design is yours and what part is borrowed from the image file.

FAQ

The API checker says that I need to make my methods `private`. Use the access modifier `private` instead of `public` in the method signature. A public method can be called directly in another class; a private method cannot. The only public method that you should have in `Art.java` is `main()`.

What will cause me to lose points on the artistic part? We consider three things: the structure of the code; the structure of the recursive function-call tree; and the art itself.

For example, the Quadricross looks very different from the in-class examples, but the code to generate it looks extremely similar to `HTree`, so it is a bad choice. On the other hand, even though the Sierpinski curve eventually generates something that looks like the Sierpinski triangle, the code is very different (probably including an angle argument in the recursive method), and so it would earn full marks.

Submission

Before submitting the assignment, please ensure you have read the COS 126 Style Guide and that your code follows our conventions. Style is an important component of writing code, and not following guidelines will result in deductions.

Note that, as part of this assignment, we may anonymously publish your art. If you object, please indicate so in your `readme.txt` when asked. We also reserve the right to pull any art, at any time, for whatever reason—and by submitting your assignment, you implicitly agree with this policy.

Submit `Transform2D.java`, `Sierpinski.java`, `Art.java` (and optional image files), and a completed `readme.txt`.

Enrichment

Fractals in the wild. Here's a Sierpinski triangle in polymer clay, a Sierpinski carpet cookie, a fractal pizza, and a Sierpinski hamantaschen.

Fractal dimension (optional diversion). In grade school, you learn that the dimension of a line segment is 1, the dimension of a square is 2, and the dimension of a cube is 3. But you probably didn't learn what is really meant by the term *dimension*. How can we express what it means mathematically or computationally? Formally, we can define the *Hausdorff dimension* or *similarity dimension* of a self-similar figure by partitioning the figure into a number of self-similar pieces of smaller size. We define the dimension to be the log (# self similar pieces) / log (scaling factor in each spatial direction). For example, we can decompose the unit square into four smaller squares, each of side length 1/2; or we can decompose it into 25 squares, each of side length 1/5. Here, the number of self-similar pieces is 4 (or 25) and the scaling factor is 2 (or 5). Thus, the dimension of a square is 2, since log (4) / log(2) = log (25) / log (5) = 2. Furthermore, we can decompose the unit cube into 8 cubes, each of side length 1/2; or we can decompose it into 125 cubes, each of side length 1/5. Therefore, the dimension of a cube is log(8) / log (2) = log(125) / log(5) = 3.

We can also apply this definition directly to the (set of white points in) Sierpinski triangle. We can decompose the unit Sierpinski triangle into three Sierpinski triangles, each of side length 1/2. Thus, the dimension of a Sierpinski triangle is log (3) / log (2) ≈ 1.585. Its dimension is fractional—more than a line segment, but less than a square! With Euclidean geometry, the dimension is always an integer; with fractal geometry, it can be something in between. Fractals are similar to many physical objects; for example, the coastline of Britain resembles a fractal; its fractal dimension has been measured to be approximately 1.25.