# ***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.

# 

# ***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 };  StdDraw.polygon(x, y); | StdDraw and polygon |
| --- | --- |

**Transform2D.java**

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

| **public class Transform2D {**  // 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)**  // 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)**  // 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 assume the arguments are in radians. Use Math.toRadians() to convert from degrees to radians.
* The transformation methods scale(), translate() and rotate() *mutate* the polygons, while copy() returns a new polygon.
* The main method **must** test each method of the Transform2D library. In other words, you must call each Transform2D method from the 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.
* t
* 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:

| // Set the x- and y-scale  StdDraw.setScale(-5.0, 5.0);  // Original polygon  double[] x = { 0, 1, 1, 0 };  double[] y = { 0, 0, 2, 1 };  // Copy of original polygon  double[] cx = Transform2D.copy(x);  double[] cy = Transform2D.copy(y);  // Rotate, translate and draw the copy  Transform2D.rotate(cx, cy, -45.0);  Transform2D.translate(cx, cy, 1.0, 2.0);  StdDraw.setPenColor(StdDraw.BLUE);  StdDraw.polygon(cx, cy);  // Draw the original polygon  StdDraw.setPenColor(StdDraw.RED);  StdDraw.polygon(x, y); |  |
| --- | --- |

## **scale()**

*Scales* the coordinates of each vertex (*xi* , *yi*) by a factor α.

* *xi* ′ = α*xi*
* *yi* ′ = α*yi*

An example 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) {  StdDraw.setScale(-5.0, +5.0);  double[] x = { 0, 1, 1, 0 };  double[] y = { 0, 0, 2, 1 };    StdDraw.setPenColor(StdDraw.RED);  StdDraw.polygon(x, y);  **scale(x, y, 2.0);**  StdDraw.setPenColor(StdDraw.BLUE);  StdDraw.polygon(x, y);  } |  |
| --- | --- |

## **translate()**

*Translates* each vertex (*xi* , *yi*) by a given offset (*dx*, *dy*).

* *xi* ′ = *xi* + *dx*
* *yi* ′ = *yi* + *dy*

An example 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) {  StdDraw.setScale(-5.0, +5.0);  double[] x = { 0, 1, 1, 0 };  double[] y = { 0, 0, 2, 1 };    StdDraw.setPenColor(StdDraw.RED);  StdDraw.polygon(x, y);  **translate(x, y, 2.0, 1.0);**  StdDraw.setPenColor(StdDraw.BLUE);  StdDraw.polygon(x, y);  } |  |
| --- | --- |

## **rotate()**

*Rotates* each vertex (*xi* , *yi*) by *θ* degrees counterclockwise, around the origin.

* *xi* ′ = *xi* cos⁡θ − *yi* sin⁡θ
* *yi* ′ = *yi* cos⁡θ + *xi* sin⁡θ

Note in the equations *xi* ′ and *yi* ′ depend on the *xi* and *yi* , 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 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) {  StdDraw.setScale(-5.0, +5.0);  double[] x = { 0, 1, 1, 0 };  double[] y = { 0, 0, 2, 1 };  StdDraw.setPenColor(StdDraw.RED);  StdDraw.polygon(x, y);  **rotate(x, y, 45.0);**  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. For example:

| // Original polygon  double[] x = { 1, 2, 2, 1 };  double[] y = { 1, 1, 3, 2 };  StdDraw.setPenColor(StdDraw.RED);  StdDraw.polygon(x, y);  // Rotate polygon  // 90 degrees counterclockwise  Transform2D.rotate(x, y, 90.0);  StdDraw.setPenColor(StdDraw.BLUE);  StdDraw.polygon(x, y); |  |
| --- | --- |

# ***Part II - Sierpinski Triangle***

The Sierpinski triangle[[1]](#footnote-0) is an example of a fractal pattern like the H-tree pattern from Section 2.3 of the textbook.

| Sierpinski triangle of order 1 | Sierpinski triangle of order 2 | Sierpinski triangle of order 3 |
| --- | --- | --- |
| *Order 1* | *Order 2* | *Order 3* |
| Sierpinski triangle of order 4 | Sierpinski triangle of order 5 | Sierpinski triangle of order 6 |
| *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.

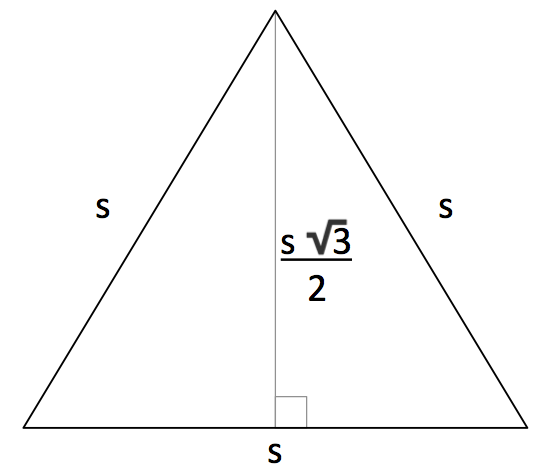
Your main task is to write a recursive function sierpinski() that plots a Sierpinski triangle of order n to standard drawing. Think recursively: sierpinski() should draw one filled equilateral triangle (pointed downwards) and then call itself recursively three times (with an appropriate stopping condition). 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**

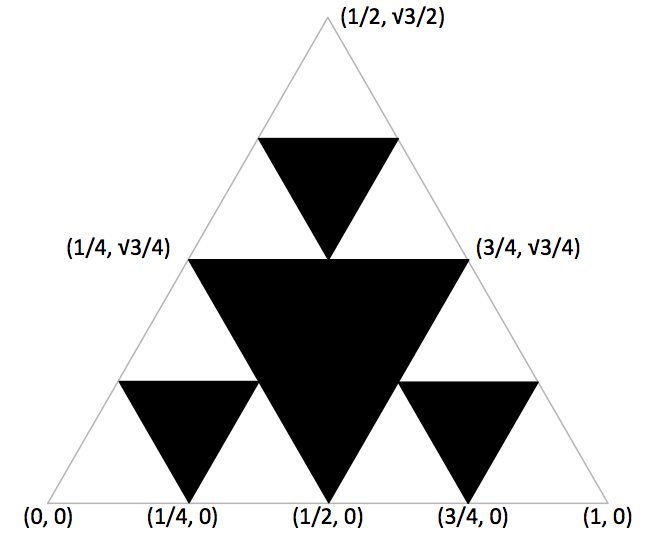
When writing your program, exercise modular design by organizing it into four functions, as specified in the following API:

| **public class Sierpinski {**  // Height of an equilateral triangle whose sides are of the specified length.  **public static double height(double length)**  // Draws a filled equilateral triangle whose bottom vertex is (x, y)  // of the specified side length.  **public static void filledTriangle(double x, double y, double length)**  // Draws a Sierpinski triangle of order n, such that the largest filled  // triangle has bottom vertex (x, y) and sides of the specified length.  **public static void sierpinski(int n, double x, double y, double length)**  // Takes an integer command-line argument n;  // draws the outline of an equilateral triangle (pointed upwards) 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 snugly inside the outline.  **public static void main(String[] args)**  **}** |
| --- |

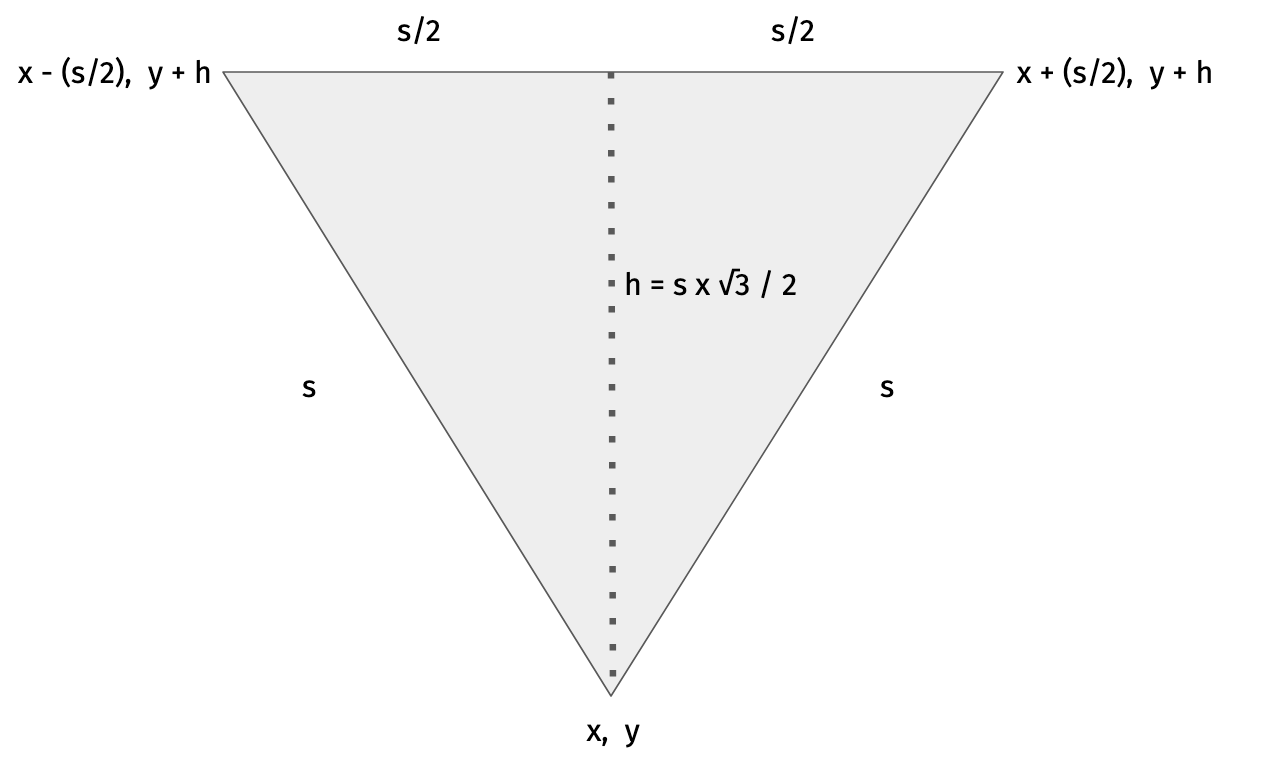
The formula for the *height* of an equilateral triangle of side length s is



Here is the layout of the initial equilateral triangle. The top vertex lies at



Here is the layout of an inverted equilateral triangle



## ***Requirements***

# To draw a filled equilateral triangle you should call the method StdDraw.filledPolygon() with appropriate arguments.

# You **must not** call StdDraw.save(), StdDraw.setCanvasSize(), StdDraw.setXscale(), StdDraw.setYscale(), or StdDraw.setScale(). These method calls interfere with grading.

1. 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](https://introcs.cs.princeton.edu/23recursion/Htree.java.html) from the textbook and lecture.
* Write the (non-recursive) function height() that takes the length of the sides in 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?
* Write a (nonrecursive) function filledTriangle() that takes three real-valued arguments (x, y, length), and draws a filled equilateral triangle (pointed downward) with bottom vertex at (x, y) of the specified side length.  
     
  To debug and 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 bottom vertex (x, y) and the specified side length. 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 this 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](https://www.cs.princeton.edu/courses/archive/fall20/cos126/assignments/sierpinski/sierpinski-part1.txt) 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.
  + 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 triangle has side length 0.5. Each successive level of recursion halves the length. Ignoring whitespace, your function should produce the [following output](https://www.cs.princeton.edu/courses/archive/fall20/cos126/assignments/sierpinski/sierpinski-part2.txt).
  + Modify sierpinski() so that it takes **four (4) arguments (n, x, y, and length)** and plots a Sierpinski triangle of order n, whose largest triangle has bottom vertex (x, y) and the specified side length. Start by drawing Sierpinski triangles with pencil and paper.
  + Remove all print statements before submitting to CodePost.

* 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](https://introcs.cs.princeton.edu/23recursion). Here are some of our favorite [student s ubmissions from a previous year](https://www.cs.princeton.edu/courses/archive/fall15/cos126/art/index.php). See also the Famous Fractals in [Fractals Unleashed](https://wayback.archive-it.org/3635/20130719033956/http://library.thinkquest.org/26242/full/index.html) for some ideas. Here is a [list of fractals, by Hausdorff dimension](https://en.wikipedia.org/wiki/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 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 7.
3. Your drawing can be a geometric pattern, a random construction, or anything else that takes advantage of recursive functions.
4. 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.
5. Your program must be organized into at least three separate functions, including main(). All functions except main() must be private.
6. You may not change the size of the drawing window (but you may change the scale). Do not add sound.
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). You must do **at least two** of these things to get full credit on Art.java (and doing more *may* yield a small amount of extra credit):
   1. call one or more Transform2D method
   2. use different parameters than our examples: f(n, x, y, size)
   3. use different StdDraw methods than in examples (e.g., ellipses, arcs, text)
   4. number of recursive calls not constant per level (e.g., conditional recursion)
   5. mutually recursive methods
   6. multiple recursive methods
   7. doesn't always recur from level n to level n-1
   8. draw between recursive calls, not just before or after all recursive calls
   9. 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](https://introcs.cs.princeton.edu/java/23recursion/Factorial.java.html)). 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.

**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 by a method in another class; a private method cannot. The only public method that you should have in Art 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](https://en.wikipedia.org/wiki/File:Quadriccross.gif) 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](https://www.robertdickau.com/sierpinskiarrow.html) 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***

# Submit Transform2D.java, Sierpinski.java, Art.java (and optional files) and readme.txt.

# ***Enrichment***

**Fractals in the wild.** Here's a Sierpinski triangle in [polymer clay](https://www.evilmadscientist.com/article.php/fimofractals), a [Sierpinski carpet cookie,](https://www.evilmadscientist.com/article.php/fractalcookies) a [fractal pizza](https://slice.seriouseats.com/archives/2010/09/john-riepenhoffs-recursive-pizza.html), and a [Sierpinski hamantaschen](https://seattlelocalfood.com/2011/03/20/sierpinski-hamantaschen-sierpinskitaschen/).

**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. 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.

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. [↑](#footnote-ref-0)