Turtle Fractal Lab

In this lab, you will be using the APCSTurtleLib to create programs which draw fractals using recursion. Recall from last semester that a Turtle is an object which can be given commands to move around the plane. As it moves, draws a line along its path. Here are the interesting parts of the Turtle class:

public class Turtle {

// creates a turtle in a 800x600 window

public static Turtle CreateTurtle() { . . . } // the turtle starts at (400, 300)

// the turtle starts at (x, y)

public static Turtle CreateTurtle( double x, double y ) { . . . }

public Turtle() { . . . } // creates a turtle starting at (400, 300)

public Turtle( double x, double y ) { . . . } // creates a turtle starting at (x, y)

public void forward( double d ) { . . . } // moves the turtle forward d pixels

public void right( double a ) { . . . } // turns the turtle to the right a degrees

public void left( double a ) { . . . } // turns the turtle to the left a degrees

// moves the turtle in a clockwise-circular arc of radius r and angle a

public void arcRight( double a, double r ) { . . . }

// moves the turtle in a counter-clockwise-circlular arc of radius r and angle a

public void arcLeft( double a, double r ) { . . . }

public void setColor( Color col ) { . . . } // changes the turtle’s drawing color

}

The static functions CreateTurtle() and CreateTurtle( double x, double y ) allow a quick creation of a turtle in it’s own window. Alternatively, you can use the constructor and the normal method of creating turtles and windows on your own. Here is a simple program using a turtle to draw a regular pentagon and its result:

public static void main( String [] args ) {

Turtle shelldon = CreateTurtle();

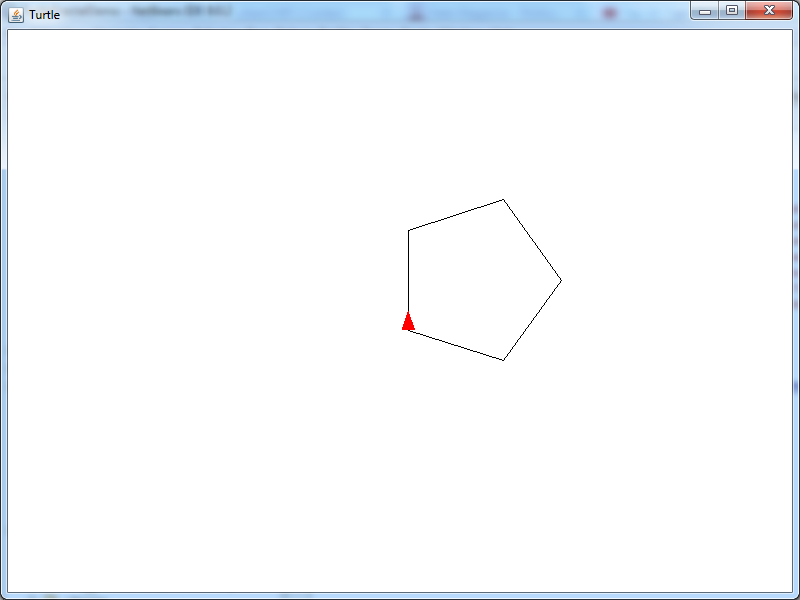
for ( int i = 0; i < 5; i++ ) {

shelldon.forward( 100 );

shelldon.right( 72 );

}

}

And the result:  


Functions Review

Suppose that you have a function which consumes a Turtle and length and causes the turtle to draw a regular pentagon. For example:

// Causes the turtle to draw a regular pentagon with given side length, turning to the right.

// PostCondition: The turtle will be in the same position and direction as it was before the function was

// called.

public static void drawPentagon(Turtle t, length l ) {

for ( int i = 0; i < 5; i++ ) {

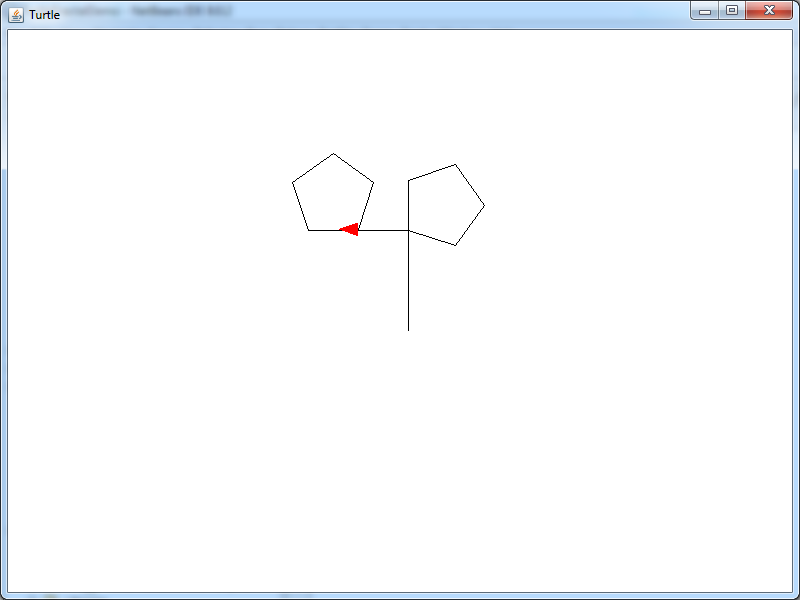
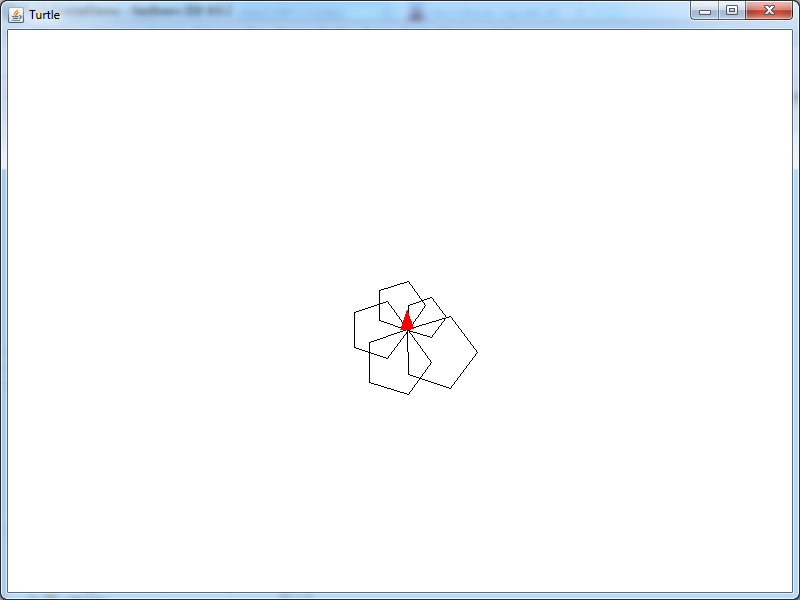
t.forward( l );

t.right( 72 );

}

}

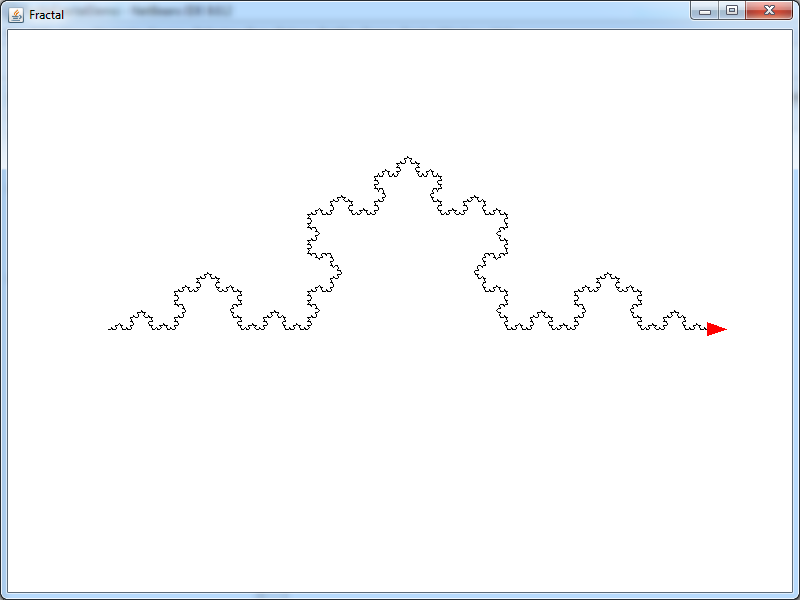
How could you use this function to draw this picture? Or this one:

The key to these programs is thinking something like “Ok, my turtle is going to start here. I want a pentagon there. To acccomplish that, I have to get the turtle to this position, then call drawPentagon with the turtle and the right size. Now I want another pentagon over there. I know that when I’m done with the first pentagon, I’ll be in this position, so I need to execute several moves/turns to get the turtle to the second location, the call drawPentagon again. Etc.” Notice that you don’t really care about the details of drawPentagon, only where the pentagon will get drawn relative to the turtle’s current position and where the turtle will be AFTER the pentagon is drawn so that you can get the turtle into position for the next task.

Drawing Fractals

The picture below is called the Koch Snowflake. This one is 600 pixels long:



Here is the code to draw this picture:

TurtleSlave t = setUpTurtleSlave();

if ( t != null )

{

t.setColor( Color.WHITE );

t.left( 90 );

t.forward( 300 );

t.right( 180 );

t.setColor( Color.BLACK );

kochSnowflake( t, 600 );

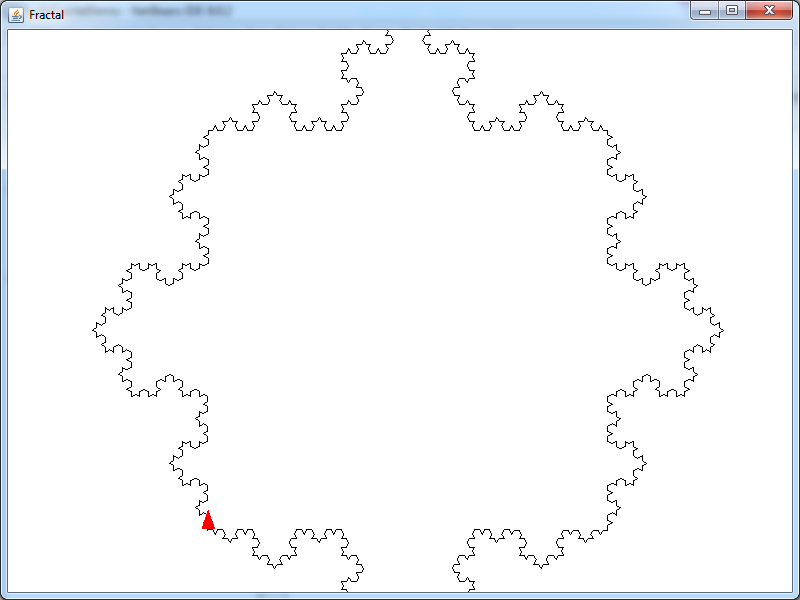
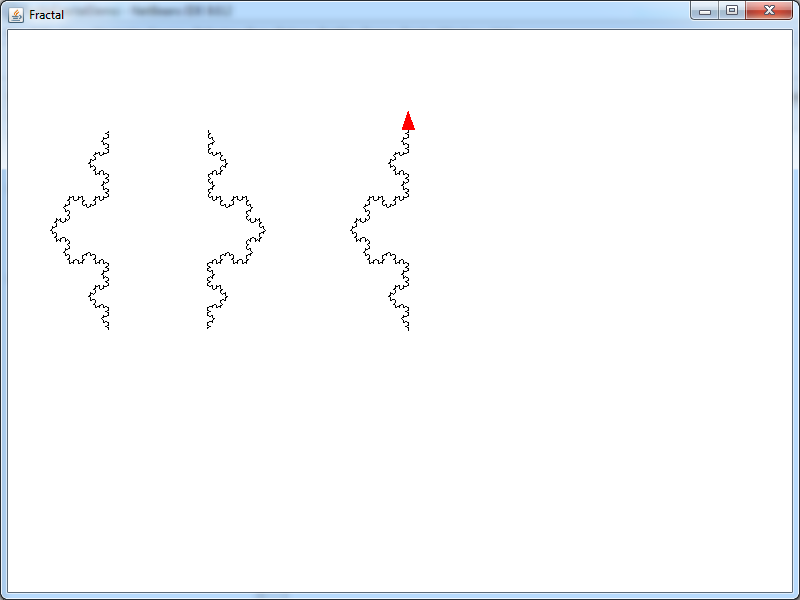
}

The first line creates a TurtleSlave (this is a fancy class, but as far as you need to be concerned, it works just like Turtle. In fact, it extends class Turtle). The next 5 lines are positioning the turtle at the far left corner of the snowflake, facing right. Then the function kochSnowflake is called with the turtle and the size of the snowflake that we want. The things to note about this function are that:

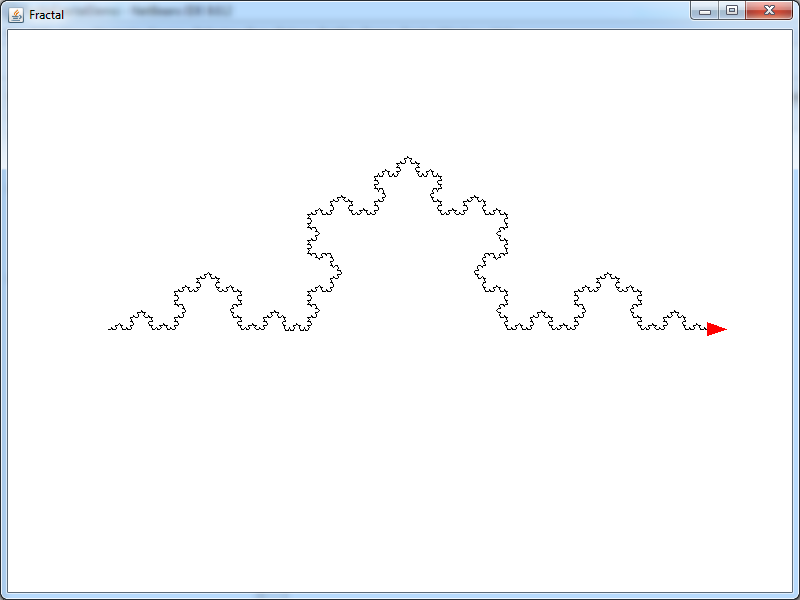
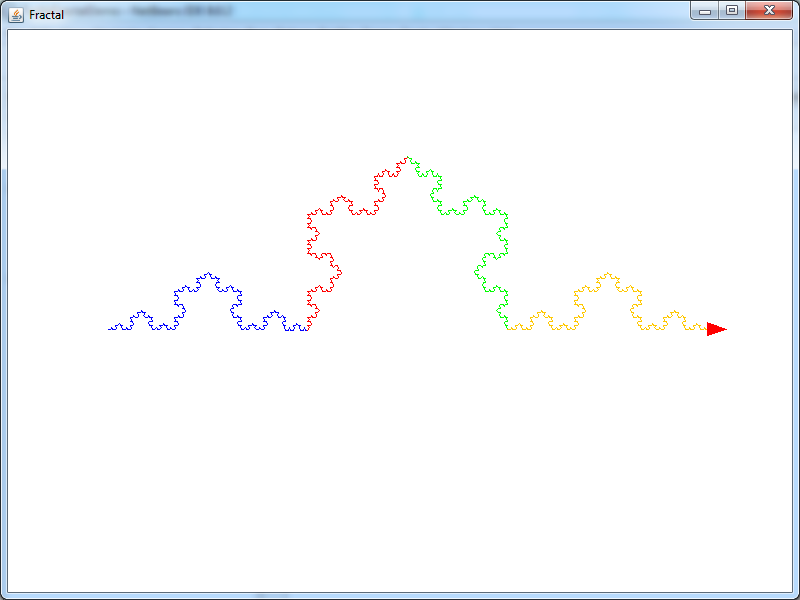
1. To use it, you need to position the turtle in the far-left corner of the figure, facing in the direction that you want the snowflake to be drawn.
2. You can specify how big the snowflake should be (and which turtle to use to draw it)
3. When the function has finished, the turtle will be at the far right corner of the figure, facing in the same direction in which it started.

Practice Exercise:

Modify the program to create the pictures shown below:



Now, you may have noticed that the Koch Snowflake consists of 4 smaller Koch Snowflakes! That is, the Koch Snowflake contains within it four smaller copies of itself, each 1/3 the size of the whole. We call such a structure a “Recursive Structure”:

Practice: Modify your program to draw the colored picture shown above. When you are done, scroll down to the next page (don’t peak until you’ve finished this practice!)

Here is the code to draw the black version (I’ve removed the code that does the initial positioning of the turtle at the far left of the window from both snippets):

t.setColor( Color.BLACK );

kochSnowflake( t, 600 );

And here’s the code that generates the color verion:

t.setColor( Color.BLUE );

kochSnowflake( t, 200 );

t.left( 60 );

t.setColor( Color.RED );

kochSnowflake( t, 200 );

t.right( 120 );

t.setColor( Color.GREEN );

kochSnowflake( t, 200 );

t.left( 60 );

t.setColor( Color.ORANGE );

kochSnowflake( t, 200 );

We can see that the path of the turtle (minus the colors) is the same in both cases. In other words, the line

kochSnowflake( t, 600 );

Is exactly equivalent to

kochSnowflake( t, 200 );

t.left( 60 );

kochSnowflake( t, 200 );

t.right( 120 );

kochSnowflake( t, 200 );

t.left( 60 );

kochSnowflake( t, 200 );

Given that we want a snowflake of size 600, how did you know to break it into snowflakes of size 200? (Why 200?). What about the positioning code in between calls to the snowflake function? How did you know to turn left 60 degrees etc? If I asked you to draw a snowflake of size N, you could call kochSnowflake( t, N ) or, equivalently, what could you do?

Take a look at the body of kochSnowflake:

public static void kochSnowflake( TurtleSlave t, double s ) {

kochSnowflakeComplete( t, s );

}

Function kochSnowflakeComplete is another function which does the hard work of drawing the snowflake. kochSnowflake just calls kochSnowflakeComplete. Let’s re-write kochSnowflake to make 4 calls to kochSnowflakeComplete so that the result is the same. Do this before you go to the next page.

Here’s the solution:

public static void kochSnowflake( TurtleSlave t, double s ) {

kochSnowflakeComplete( t, s/2 );

t.left( 60 );

kochSnowflakeComplete( t, s/2 );

t.right( 120 );

kochSnowflakeComplete( t, s/2 );

t.left( 60 );

kochSnowflakeComplete( t, s/2 );

}

The point of kochSnowflakeComplete is that it does all the hard work for us. It’s job is to draw a snowflake of a given size. That seems like a really complicated thing to do! What does kochSnowflake do? (not the details about calling kochSnowflakeComplete… what does kochSnowflake do?)

Right! kochSnowflake also draws a snowflake! It just does it in a different way than kochSnowflakeComplete. What would happen if we replaced all the calls to kochSnowflakeComplete with calls to kochSnowflake:

public static void kochSnowflake( TurtleSlave t, double s ) {

kochSnowflake( t, s/2 );

t.left( 60 );

kochSnowflake( t, s/2 );

t.right( 120 );

kochSnowflake( t, s/2 );

t.left( 60 );

kochSnowflake( t, s/2 );

}

Try it! What happened? What went wrong?

What did we just do? A snowflake is made up of 4 smaller snowflakes. What happens when we draw the smaller snowflakes? The program breaks them up into even smaller snowflakes. Which get reduced to 4 tiny snowflakes each of which… And so on... When does it stop? That’s the problem… Our program doesn’t know when to stop breaking down.

Mathematically speaking, drawing a snowflake is equivalent to drawing 4 smaller snowflakes, but practically speaking, we need to decide that for some tiny size, we’ll just approximate the snowflake with something simple. So we’ll start by checking the size of the snowflake before we break down into smaller parts. If the size is still big enough, we’ll break down into smaller sizes:

public static void kochSnowflake( TurtleSlave t, double s ) {

// if the size is big enough, split into smaller snowflakes

if ( s > 5 ) {

kochSnowflake( t, s/2 );

t.left( 60 );

kochSnowflake( t, s/2 );

t.right( 120 );

kochSnowflake( t, s/2 );

t.left( 60 );

kochSnowflake( t, s/2 );

}

}

Now try it… what went wrong this time? When we get down to too small a size, the program does … nothing!

What should we do when the size is too small (say s = 3 )? A hint can be found in what the function promises to do. The function promises that after the function is done, the snowflake will be drawn, but the turtle will be at the far end of the snowflake. How far away is that? What (simple) command could we give the turtle in order to make this true?

Here’s the new function:

public static void kochSnowflake( TurtleSlave t, double s ) {

// if the size is big enough, split into smaller snowflakes

if ( s > 5 ) {

kochSnowflake( t, s/2 );

t.left( 60 );

kochSnowflake( t, s/2 );

t.right( 120 );

kochSnowflake( t, s/2 );

t.left( 60 );

kochSnowflake( t, s/2 );

}

else {

t.forward( s );

}

}

Try your program now.

Exercises: Experiment with the other fractal functions in the program. After you’ve experimented, re-write them so that they call themselves rather than the xxxComplete function to do their work.