Homework 5 :: Classes, Interfaces, and Fractals

15-121 Fall 2015
Due: October 13, 2015 (11:50 pm)
Basecode, Hand-In

Overview

In this assignment, you will be drawing and analyzing fractals. If you are unfamiliar with fractals, you are encouraged to read the Wikipedia article on fractals, as it may be helpful in your development. In addition, you will be implementing Java Classes and Java Interfaces.

As with all assignments, you will be graded in part on your coding style. Your code should be easy to read, organized, and well-documented. Be consistent in your use of indentation and braces. See the style guide for more details..

Background :: Code

`Fractals.java` — This file contains the `main` method, which you are welcome to use for your testing. There is one additional method in this file, which you will complete near the end of the assignment.

`Curve.java` and `Fractal.java` — You will be asked to complete these interfaces in the assignment.

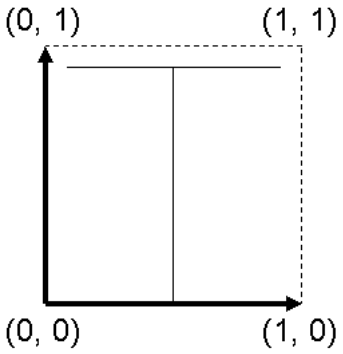
`SketchPad.java` — You will use this class to draw simple pictures (and eventually, complex fractals). Here's an example of how to use the SketchPad:

```
SketchPad() // creates a new Window
void drawLine(double x1, double y1, double x2, double y2) // draws a line from (x1, y1) to (x2, y2)
```

`drawLine`'s argument values must fall in the range from 0 to 1:

```
SketchPad pad = new SketchPad();
pad.drawLine(0.1, 0.9, 0.9, 0.9);
pad.drawLine(0.5, 0, 0.5, 0.9);
```

This will produce the following:



Exercise :: Identify

Download the Basecode and fill in the required fields like so:

```
/**
 * @author [First Name] [Last Name] <[Andrew ID]>
 * @section [Section Letter]
 */

/**
 * @author Jess Virdo <jvirdo>
 * @section A
 */
```

Exercise :: Explore

Use the `SketchPad` to draw a very simple line drawing. Don't spend too much time on this part — just enough to get the hang of the `SketchPad` class. If you like, you may write your drawing code in the main method in `Fractals.java`. You should not leave any exploratory code in your files.

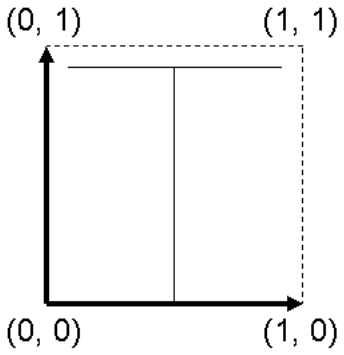
Exercise :: `Line` Class

Create a new class called `Line`. (Recall that you must name the file `Line.java`). The `Line` class should be implemented so that is can be used as follows:

```
SketchPad pad = new SketchPad();
Line line1 = new Line(0.1, 0.9, 0.9, 0.9);
Line line2 = new Line(0.5, 0, 0.5, 0.9);
```

```
line1.draw(pad);
line2.draw(pad);
```

This will produce the same result as in the overview:



You should thoroughly test this method before continuing.

Exercise :: `Curve` Class

In addition to lines, we might imagine creating classes for drawing circles, rectangles, etc. — all of which would need a `draw` method. All these hypothetical classes would share the same interface, but each would implement the `draw` method differently, so that the correct shape would be drawn.

Modify the `Curve` interface (in `Curve.java`), so that it requires the `draw` method be implemented (by all classes that implement this interface). You will also need to modify the `Line` class.

When you've finished this task, you should be able to have a variable of type `Curve` refer to an instance of the `Line` class. Likewise, you should be able to invoke the `draw` method on a `Curve` variable, as shown below:

```
Curve line1 = new Line(0.1, 0.9, 0.9, 0.9);
Curve line2 = new Line(0.5, 0, 0.5, 0.9);

line1.draw(pad);
line2.draw(pad);
```

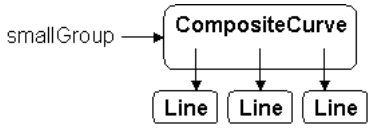
You should thoroughly test this method before continuing.

Exercise :: `CompositeCurve` Class

We could define classes for drawing rectangles, polygons, etc., but it turns out we can make a single class that can do all this! We'll call this new class `CompositeCurve`, and we'll be able to add as many lines as we want to it, and then draw them all at once:

```
CompositeCurve smallGroup = new CompositeCurve();
smallGroup.add(new Line(0.1, 0.9, 0.9, 0.9));
smallGroup.add(new Line(0.5, 0, 0.5, 0.9));
smallGroup.add(new Line(0.1, 0.9, 0.1, 0.8));
smallGroup.draw(pad);
```

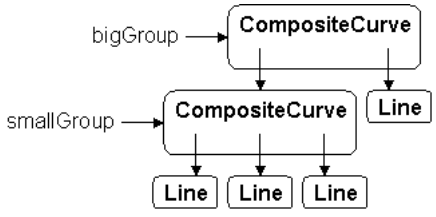
We might picture a `CompositeCurve` as follows:



Yes, we can use `CompositeCurve`'s `add` method to add `Line` objects, but why stop there? Using what we know about interfaces, it should be a cinch to use the same add method to add `CompositeCurves` to a `CompositeCurve`, as shown here:

```
CompositeCurve bigGroup = new CompositeCurve();
bigGroup.add(smallGroup);
bigGroup.add(new Line(0.9, 0.9, 0.9, 0.8));
bigGroup.draw(pad);
```

Now, drawing `bigGroup` will make four lines appear — the one we just added to `bigGroup`, and the three lines we already added to `smallGroup`. We can picture this as follows:

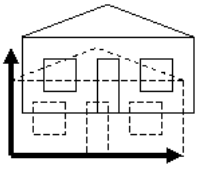


Create the `CompositeCurve` class. Do not create more than one field in the `CompositeCurve` class!

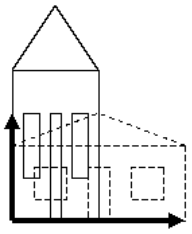
You should thoroughly test this method before continuing.

Exercise :: Transform

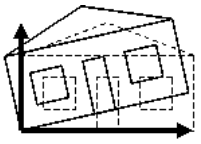
One thing we might do with a drawing is to transform it. There are three fundamental transformations: translating, scaling, and rotating. In the examples below, the original image is shown in dashed lines, and the transformed image is shown in solid lines. The lower left corner of the original image is situated at (0, 0).



house.translate(0.1, 0.4);



house.scale(0.5, 2);



house.rotate(15);

Note that scaling is performed relative to the origin (0, 0). For example, if a line originally connects the point (0.2, 0.2) to (0.4, 0.4), then, after scaling the x-coordinates by a factor of 0.5 and the y-coordinates by a factor of 2, the line will now connect the point (0.1, 0.4) to (0.2, 0.8). Similarly, we will rotate our lines about the origin.

Modify your Curve interface so that it requires classes to implement the following three methods, each of which modifies the curve in question:

```
void translate(double tx, double ty) // translates to the right by tx and up by ty
void scale(double sx, double sy) // scales width by sx and height by of sy
void rotate(double degrees) // rotates counter-clockwise by degrees (about the origin)
```

Of course, this means you'll need to implement each of these methods in your Line and CompositeCurve classes. You should be able to figure out all of these on your own, except for rotating a line. If one of the endpoints of your line is (x, y), and the line is being rotated by 胃 degrees, then the new location of the endpoint (x', y') is given by the following formula. You will need to perform this calculation for each endpoint. (Remember that you're rotating about the origin — not about one of the end points.)

```
x' = x cos θ - y sin θ
y' = x sin θ + y cos θ
```

Check your logic. Implementing this formula incorrectly will be the source of much misery on this assignment! You will need the following methods in Java's Math class to implement this formula in Java:

```
static double sin(double radians)
static double cos(double radians)
static double toRadians(double degrees)
```

When you've finished implementing the modified Curve interface, make sure to test all your new methods. Don't go on to the next exercise until you're certain these work correctly!

Exercise :: copy

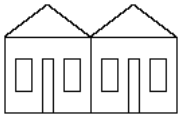
Modify your Curve interface again so that it requires classes to implement the copy method:

```
Curve copy() // returns a deep copy of this Curve
```

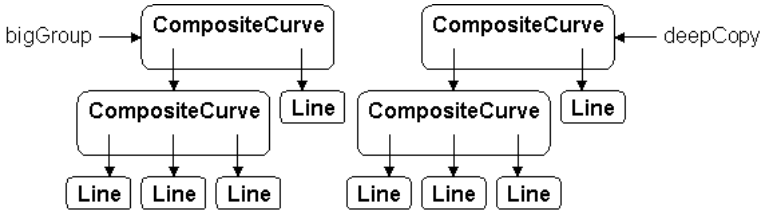
The Curve returned by copy should be exactly like the original Curve, but completely independent from it. In other words, if we translate/scale/rotate the original Curve, it should not affect the copy (and vice versa). For example, if house is originally associated with a Curve whose base stretches from (0, 0) to (0.5, 0), then the following code:

```
Curve house2 = house.copy();
house.translate(0.5, 0);
house.draw(pad);
house2.draw(pad);
```

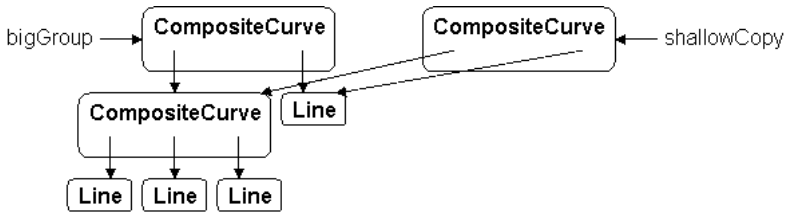
should result in the image below (even if you have not previously drawn house):



This requires that CompositeCurve's copy method return a deep copy, like the one shown below:



By contrast, we don't want the copy method to produce a shallow copy, like the one shown below:

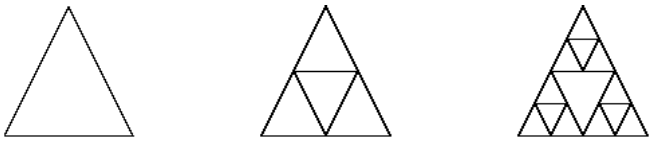


Go ahead and implement `Line`'s `copy` method, so that it returns a new `Line` at the same coordinates. Likewise, implement `CompositeCurve`'s `copy` method, so that it returns a new `CompositeCurve`, representing a deep copy of the original curve.

Be sure to test your `copy` methods. Don't go on to the next exercise until you're certain these work correctly!

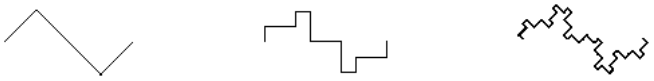
Exercise :: Fractals

Fractals are recursive drawings (although that doesn't mean your code needs to be recursive). You can create a fractal by creating an initial image; then shrinking, copying, and transforming it to create a new image; and then shrinking, copying, and transforming it again according to the same rule; and so on. The following sequences of pictures show the first few iterations of some popular fractals:



Step 0, Step 1, and Step2 of the Sierpinski Triangle

Step 0, Step 1, and Step2 of the Koch's Snowflake



Step 0, Step 1, and Step2 of the Coastline fractal

Notice that, each step consists of several smaller copies of the previous step. For example, each step of the Sierpinski fractal consists of 3 copies of the previous step. (Don't think of each step as adding more and more smaller and smaller triangles!) Because all fractals are generated in largely the same manner, we will use the `Fractal` interface to capture this idea. Modify the `Fractal` interface (in `Fractal.java`) so that it requires the following two methods:

```
Curve step0() // returns a new Curve representing step 0 of the fractal
Curve transform(Curve curve) // given a Curve representing step n of the fractal,
                                                                    // uses it to return a new Curve repr
                                                                    // n+1 of the fractal
```

Pick one of the three fractals shown above, and create a class called `Sierpinski`, `Koch`, or `Coastline` that implements the `Fractal` interface. In that class, implement the rules that correspond to that fractal.

Your `transform` method should take in a `Curve` representing some iteration of the fractal you chose, and return a `Curve` representing the next iteration of that fractal.

For example, the Sierpinski Triangle requires that you shrink the previous step and assemble three copies of it. (Note that scaling and rotating are relative to the origin. If you are scaling and rotating, you'll want to move your curve to the origin first, then scale and/or rotate, and then move it to its final location.)

Your code should not actually draw the fractal — it should only create a `Curve` that we might choose to draw later. Test that you can use your new class to draw step 0 for your fractal. Next test that you can use the `transform` method to draw step 1, and that you can transform that to draw step 2.

You get to decide how big to make your fractal, and where to place it (as long as it would be visible in a `SketchPad`), but the `Curve` returned by transform should always be exactly the same size (and in the same place) as the `Curve` that `transform` took in. Likewise, the `Curve` returned by `transform` should be exactly the same size (and in the same place) as the `Curve` returned by step 0. (You may find it easiest to write the code if you make sure that your `Fractal` always fills the region from `(0, 0)` to `(1, 1)`).

Exercise :: generateFractal

Complete the `generateFractal` method in `Fractals.java`, which should return a `Curve` corresponding to the given step of the given `Fractal`. For example, if you pass a `Sierpinski Fractal` and a step value of 2, then you should get back a `Curve` corresponding to step 2 of the `Sierpinski Fractal`, as shown earlier. Now test that you can use this method to draw your fractal.

Exercise :: Do it Again

Finally, pick one of the two fractals that you did not already implement, and go ahead and implement it in a class named `Sierpinski`, `Koch`, or `Coastline`. You should be able to use the `generateFractal` method (without changing it!) to draw your second `Fractal`. You do not need to implement a third fractal.

Submitting your Work

When you have completed the assignment and tested your code thoroughly, create a `.zip` file on your work. Your `.zip` file must

include seven (7) files:

1. CompositeCurve.java
2. Curve.java
3. Fractal.java
4. Fractals.java
5. Line.java
6. One fractal from the following three: Sierpinski.java, or Koch.java, or Coastline.java
7. A second fractal from the following three: Sierpinski.java, or Koch.java, or Coastline.java

Do not include any .jar or .class files when you submit, and zip only the files listed above. Do not zip not an entire folder containing the file(s).

Once you've zipped your files, visit the Autolab site -- there is a link on the top of this page -- and upload your zip file.

Keep a local copy for your records.