Lab: Fractals

Background: Interface vs. Implementation

One of the most important ideas in computer science is the distinction between **interface** and **implementation**. For example, you operate a cell phone in much the same way you operate an ordinary telephone, even though the two require very different electronics. Therefore, we might say that cell phones and ordinary phones have similar interfaces but radically different implementations. Here are working definitions of these terms:

interface – the abstract way in which you interact with an object

implementation – the concrete details of how an object works

Background: Java Interfaces

In the popular Marker Game, 8 markers are placed on a table, and two players take turns removing 1 or 2 markers at a time. Whichever player removes the last marker loses.

The game has two players and an organizer. Each player has a strategy dictating how many markers to remove. Different players may implement different strategies. No matter what strategy a player uses, he/she must support the same interface—the ability to answer how many markers to remove when a certain total number of markers are left.

In Java, we use an `interface` definition to give a name to a collection of method calls that a class may implement, without indicating how those methods should be implemented. For example, we could define a `Player` interface as follows:

```
public interface Player
{
     //Precondition:   total > 1, representing the
     //                number of markers left.
     //Postcondition:  returns the number of
     //                markers to remove:  1 or 2.
     int howMany(int total);
}
```

Note that `Player` is just an interface—not an implementation. Therefore, we can't create a new `Player` and call methods on it. We first need to implement this interface.

Here are two such implementations of `Player`:

```
public class SimplePlayer implements Player
{
     public int howMany(int total) { return 1; }
}

public class SmartPlayer implements Player
{
     public int howMany(int total)
     {
          if (total % 3 == 0) return 2;
          return 1;
     }
}
```

Notice that both classes are declared to implement `Player`. This requires that each implement the `howMany` method defined in the `Player` interface. The classes may implement other methods (and even other interfaces), too, but they must implement `howMany` (and declare it as `public`). Now we can instantiate players as follows:

```
Player simpleton = new SimplePlayer();
Player genius    = new SmartPlayer();
```

We can always cast these variables back to their original types. For example, `(SmartPlayer)genius`. Thus, `Player` is more general than `SmartPlayer`, but more specific than `Object`.

Our game also has an organizer, who handles the markers and alternates asking each player how many to remove. The organizer does not need to know anything about the players' strategies. Therefore, the organizer's code should interact with objects that implement the `Player` interface—not with `SimplePlayer` and `SmartPlayer` objects. The organizer's code might therefore be implemented as follows:

```
//Precondition:  It is current’s turn, and total
//               markers remain.
//Postcondition: Returns the winning player.
public Player play(Player current, Player next,
                   int total)
{
     if (total == 0)
          return current;
     else
          return play(next, current,
                      total - current.howMany(total));
}
```

Now we can have players with different strategies compete against each other. For example:

```
play(simpleton, simpleton, 8);
play(simpleton, genius, 8);
play(genius, genius, 8);
```

Background: Etch a Sketch

In this lab, we'll draw simple pictures using the `SketchPad` class (which you'll need to download). When you make a new `SketchPad`, a drawing window will appear. Here are the messages that `SketchPad` supports.

```
//Draws a line from (x1, y1) to (x2, y2).
public void drawLine(double x1, double y1,
                     double x2, double y2)

//Sets the color to use for future lines.
public void setColor(java.awt.Color color)
```

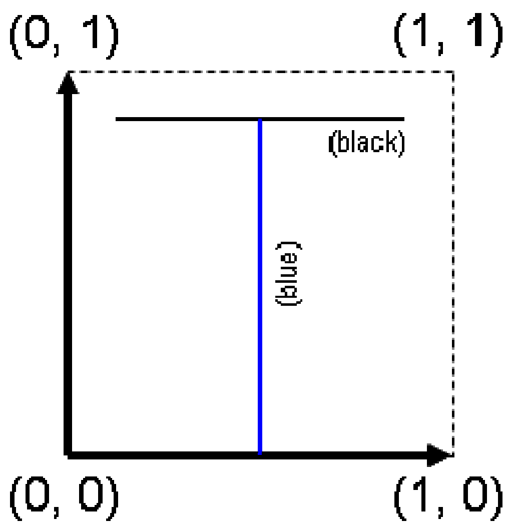
Notice that `drawLine`'s parameters are of type `double`, a primitive type that allows us to use decimal numbers. In particular, `drawLine`'s argument values must fall in the range from 0 to 1, as shown in the following example (which draws a horizontal line near the top of the view).

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

The `setColor` method uses Java's built-in `Color` class. To use it, you will need to import `java.awt.Color`. A number of `Color` constants have been defined in this class (which mysteriously do not follow Java's all-caps convention for constants). These include `black`, `blue`, `cyan`, `gray`, `green`, `magenta`, `orange`, `pink`, `red`, `white`, and `yellow`. (The `SketchPad`'s default color is black.) The following code will change the color to blue and draw a vertical blue line in the middle of the view.

```
pad.setColor(Color.blue);
pad.drawLine(0.5, 0, 0.5, 0.9);
```

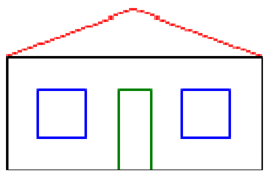
This diagram shows the `SketchPad`'s coordinate axes, along with the two lines we've drawn.

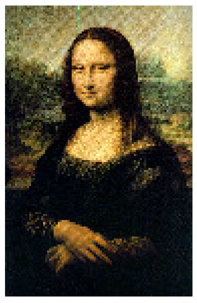


Exercise: Paint by Numbers

Write a method to draw a simple (but mildly interesting) line drawing. You may use the following images as a guideline. (Don't spend too much time on this part!)

Too Simple


Just Right


Also Good

Exercise: Power Lines

Create a new class called `Line`, which can be used as follows to draw the "T" picture shown earlier.

```
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);
pad.setColor(Color.blue);
line2.draw(pad);
```

Test your `Line` class by using it to draw the simple drawing you created in the previous exercise.

Exercise: Curve Your Enthusiasm

In addition to lines, we might create classes for drawing circles, rectangles, etc., all of which will 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. Let's therefore define a new interface in Java called `Curve`. For now, this interface will only support the `draw` method. (If you're not sure how to do this, refer to the examples at the beginning of this handout.)

When you've finished this task, you should be able to have a variable of type `Curve` point 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);
pad.setColor(Color.blue);
line2.draw(pad);
```

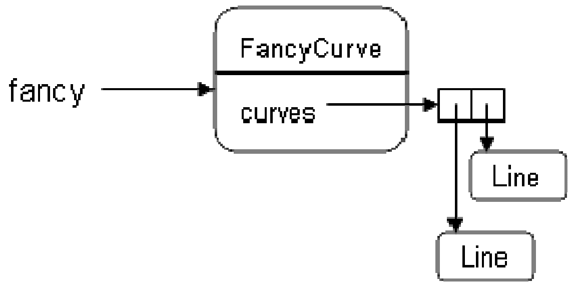
To test your interface, modify the method you wrote earlier, so that it uses variables of type `Curve` to draw your simple picture.

Exercise: Fancy Schmancy

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

```
FancyCurve fancy = new FancyCurve();
fancy.add(new Line(0.1, 0.9, 0.9, 0.9));
fancy.add(new Line(0.5, 0, 0.5, 0.9));
fancy.draw(pad);
```

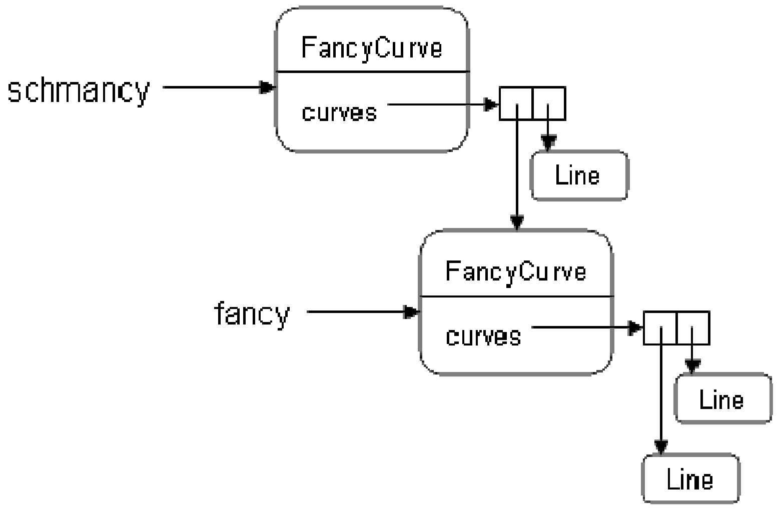
Clearly, `FancyCurve` will need to remember what lines to draw, as shown below.



We can use `FancyCurve`'s `add` method to add `Lines`, but why stop there? Using what we know about interfaces, it should be a cinch to use the *same* `add` method to add `FancyCurves` to our `FancyCurve`, as shown here.

```
FancyCurve schmancy = new FancyCurve();
schmancy.add(new Line(0.1, 0.9, 0.5, 0));
schmancy.add(fancy);
schmancy.draw(pad);
```

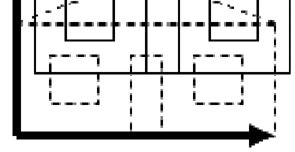
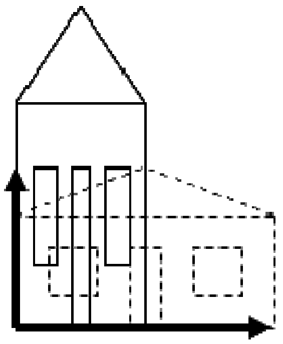
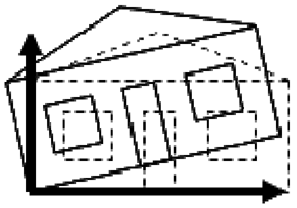
Now, drawing `schmancy` will make three lines appear—the one in the `FancyCurve` associated with `schmancy`, and the two in the `FancyCurve` associated with `fancy`.



Go ahead and create the `FancyCurve` class, making appropriate use of the `Curve` interface. Then go modify your simple drawing so that at least one part of your drawing is represented by lines added to a `FancyCurve`. Finally, add these `FancyCurve`s and any other `Line`s to a single instance of `FancyCurve`, representing the entire drawing. Make sure that asking this single `FancyCurve` to draw causes the entire picture to appear.

Exercise: Transformers

One fun thing to 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).

Translating *Scaling* *Rotating*

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

```
//translates to the right by tx and up by ty
void translate(double tx, double ty);

//scales the width by sx and the height by sy.
//scaling is performed relative to the origin.
void scale(double sx, double sy);

//rotates counter-clockwise by degrees.
void rotate(double degrees);
```

Of course, you'll need to implement each of these methods in your `Line` and `FancyCurve` classes. The trickiest case is rotating a line. If one of the endpoints of your line is (x, y), and the line is being rotated by θ 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.)

$$x' = x \cos \theta - y \sin \theta$$
$$y' = x \sin \theta + y \cos \theta$$

If you haven't seen sine, cosine, and radians in math class yet, just ask a friend for help (or ask your teacher, who also implements the `Friend` interface). Like Harker, Java comes with a `Math` class. You can use it to compute sine and cosine.

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

Note that your angle is in degrees (based on 360), but the `Math` class requires radians (based on 2π). You'll need to convert your angle to radians. You'll find the constant `Math.PI` helpful here.

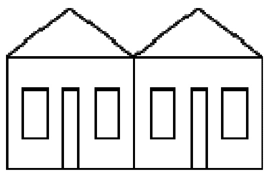
When you've finished implementing the modified `Curve` interface, try calling the `translate` method on your test drawing before calling the `draw` method. When this works correctly, try calling the `scale` method instead, and finally the `rotate` method. Don't go on to the next exercise until you're certain these work correctly!

Exercise: Copy Cat

Modify your `Curve` interface so that it requires classes to implement the `copy` method, which should take no arguments and return a `Curve`. Then go ahead and implement `Line`'s `copy` method, so that it returns a *new* `Line` at the same coordinates. Likewise, implement `FancyCurve`'s `copy` method, so that it returns a *new* `FancyCurve`, *consisting of new curves*.

```
house.scale(0.5, 1);
house2 = house.copy();
house2.translate(0.5, 0);
house.draw(pad);
house2.draw(pad);
```

If `house` is originally associated with an image whose base stretches from (0, 0) to (1, 0), then the above code should draw the following picture.

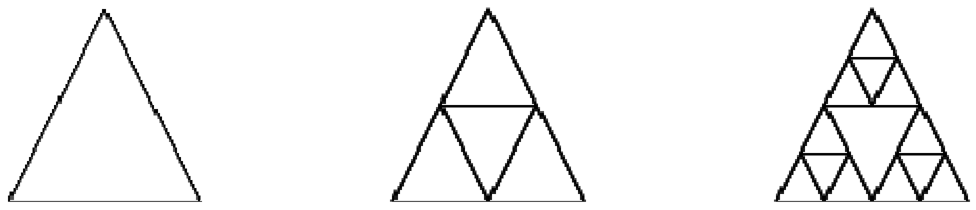


Be sure to test that your `copy` method works correctly, by (1) making a copy of the `FancyCurve` representing your test drawing, (2) transforming one of these copies differently than the other, and (3) drawing each copy.

Exercise: Fractals!

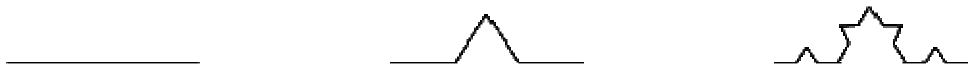
Fractals are recursive drawings (although that doesn't mean your code needs to be recursive). You can create a fractal by drawing an image, transforming and copying it to create a new image, and then repeatedly transforming and copying the resulting images. The following sequences of pictures show the first few iterations of some popular fractals.

Sierpinski Triangle



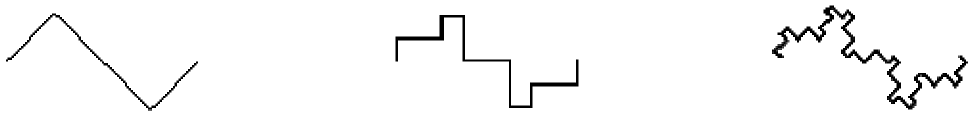
Step 0 *Step 1* *Step 2*

Koch Curve



Step 0 *Step 1* *Step 2*

Coastline Curve



Step 0 *Step 1* *Step 2*

Because all fractals are generated in largely the same manner, we'll start by declaring an interface called `FractalGenerator`, which should require the following two methods.

```
//returns a new Curve representing step 0 of the
//fractal
Curve step0();

//given a curve representing step n of the fractal,
//uses that curve to return a new curve representing
//step n+1 of the fractal.
Curve transform(Curve curve);
```

Now pick one of the fractals shown earlier. (You'll probably find the Sierpinski Triangle to be the easiest.) Create a class for generating this fractal, which will implement the `FractalGenerator` interface. *You should not need to define any instance variables.*

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 scale 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 rotate, and then move it to its final location.)

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".

Finally, in the class you've been using to test your code, implement the following method.

```
    public static Curve generateFractal
                  (FractalGenerator generator, int levels)
```

For example, if you call `generateFractal` with levels set to *i*, it should return "step *i*" of your fractal. Now use this method to draw your fractal!

Additional Credit

Additional points will be rewarded for implementing multiple `FractalGenerator`s for drawing different fractals (even making up your own fractal), and just generally drawing interesting designs. Have fun!