

CS1322 Homework 6

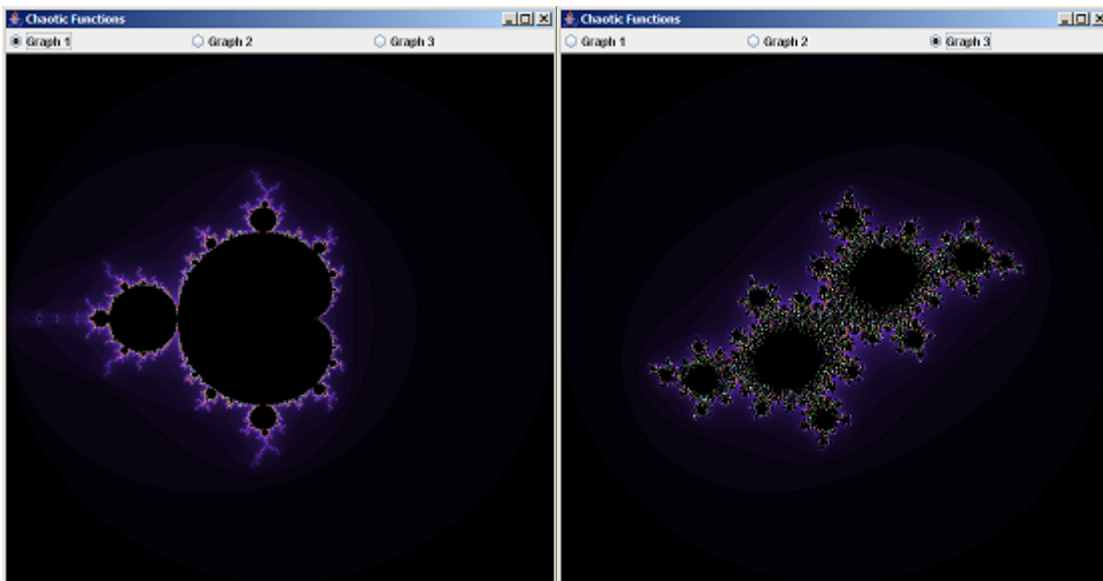
Due: Thursday, March 3 @ 6:00 pm

Chaotic Functions

(or, how Java interfaces saved me from having to deal with complex math just to draw nice pictures)

Note: Read the *whole* assignment before beginning!

For Homework 6, you are going to write a program to make graphs of chaotic functions (or, nonlinear iterating functions). What is a chaotic function, you may ask? Well, you really don't need to know to complete this assignment. This assignment is partially a demonstration of how interfaces work to hide implementation details. To give you something to look forward to, in the end you will produce pictures like these:



Before you get scared, this assignment is *not* harder than previous ones! It uses the same GUI ideas you have seen in your previous homework and in example programs in the book (Chapters 3, 4, and 5). Specifically, your program will have:

1. a class `ChaoticGraph` with a main method. That method will create a `JFrame`, as usual, and it will add a `MainPanel` to that frame.
2. a class `MainPanel`, which will extend `JPanel`. That panel will not have a `paintComponent` method: it will just do the standard drawing of a panel. Inside it, there will be three radio buttons labeled “Graph 1”, “Graph 2” and “Graph3”, as well as a different panel of type `ChaoticGraphPanel`, which will display one of the pictures above. When each radio button is clicked, it will change the chaotic function that is being displayed.
3. a class `ChaoticGraphPanel`, which handles the drawing. This class will also extend `JPanel` and it will provide its own `paintComponent` method. The `paintComponent`

method will do all the drawing of the chaotic function. The panel should be 600 by 600 pixels in size.

Let's look at these classes one-by-one:

1. ChaoticGraph class

This is very straightforward. It will just create a frame and add a panel to it. You already did this in your previous homework. If you want to see examples of this pattern, check out listings 4.5, 4.7, 5.15, 5.17, 5.19, etc. in the book. This is the standard structure we've used all along.

2. MainPanel class

This will need to have in it three radio buttons plus one more panel, of type ChaoticGraphPanel. If you need to see an example using radio buttons, check out listing 5.25 in the book. Radio buttons are similar to the plain buttons you used in your previous homework. The only difference is that you need to put them in a button group so that only one of the radio buttons can be active at a time. You need to set up a listener object to handle the events generated. The easiest thing to do is to have the listener class be a nested class of MainPanel. In the listener, you need to distinguish which button generated the event. Did we mention listing 5.25 in the book? It does exactly all the above.

When a radio button is selected, the listener should somehow notify the ChaoticGraphPanel that the function to display has changed. There are several ways to do this, but the most straightforward is to have a public method in ChaoticGraphPanel that the listener calls. That method will then change a variable (or several variables) inside the ChaoticGraphPanel so that next time the panel draws itself (in paintComponent) it draws the right function. You should then call repaint() to have your component get redrawn.

3. ChaoticGraphPanel class

This is the main workhorse of the entire application. Yet, the complex parts will be hidden behind an interface and you will only need to call the interface methods. The main idea is the following: in paintComponent, you will write a loop that goes over every pixel on the screen. For every pixel, you will call a method, which we provide pre-written, and this method will give you the color for that pixel. You will draw a pixel with that color on the screen and ... that's it!

Of course, the devil is in the details. Here are the specifics. We give you an interface:

```
public interface ChaoticFunction
{
    public int getColor(double x, double y);
    public double getDefaultXMax();
    public double getDefaultYMax();
    public double getDefaultXMin();
    public double getDefaultYMin();
}
```

```
}
```

We also give you some more code: classes that implement this interface (JuliaSet and MandelbrotSet) and a class ChaoticFunctionGenerator. Yet you do not need to look inside this code at all! You just need to call the static method `getFunction` in `ChaoticFunctionGenerator` and pass it 1, 2, or 3 depending on which of the three graphs you want to draw. (This is the information you got when the user selected one of the three radio buttons.) This function will then return an object that supports the `ChaoticFunction` interface. You can call all 5 methods of the interface on this object. For example, if you do:

```
ChaoticFunction f = ChaoticFunctionGenerator.getFunction(1);
```

then `f` will be a reference to an object (which happens to represent the Mandelbrot chaotic function) and you can call `getColor`, `getDefaultXMax`, etc. on that object.

But what do these methods do? The last four methods (`getDefault...`) tell you the maximum and minimum `x` and `y` that the function supports. Note that these are double numbers! So, for instance, you may write

```
double xMax = f.getDefaultXMax();
```

and you will get a maximum value for `x`. This means you should never pass to `getColor` an `x` argument that is larger than `xMax`!

The `getColor` method is the one that you will call on every pixel. Notice, however, that your pixels have integer `x` and `y` coordinates between 0 and 600. But, `getColor` accepts `x` and `y` coordinates that are double numbers between some `xMin` and `xMax`, and some `yMin` and `yMax`, respectively. You will need to translate between the screen coordinates and the coordinates that the chaotic function supports. This means you should scale the screen coordinates, from 0 to 600 down to the `xMin` to `xMax` (or `yMin` to `yMax`) range.

Yet one more thing to remember is that the `y` coordinates go in different directions. In a normal `XY` plane, positive on `y` is up, but in pixels, positive `y` is down. So when you do your translations, keep that in mind. If this last part confuses you, leave it for later. You'll just get the shapes upside down. E.g., see the second picture shown earlier and compare it with your program output for Graph 3. Is it roughly the same shape or is it upside down?

When you use `getColor` on a pixel, you get back an integer number. This number is a color "intensity". You can map this number to a shade of a color or to complex colors. For instance, imagine that your program says:

```
int color = f.getColor(xTrans, yTrans);
```

Then you can use this color as a shade of blue if you do:

```
g.setColor(new Color(0,0,color%256); //g is the argument to paintComponent
```

or you can draw your function in a shade of red if you do:

```
g.setColor(new Color(color%256,0,0);
```

or you can have lighter shades of blue:

```
g.setColor(new Color(0,0,(color*8)%256);
```

or you can create combinations:

```
g.setColor(new Color((color*4)%256,(color*2)%256,(color*8)%256));
```

This is all up to you. Feel free to experiment.

Once you've set the color, you put the pixel on screen with the right color by calling `g.fillRect(x,y,1,1)`. (You are filling a box, one pixel long and wide.)

Turn-in

- Turn in all the .java files needed to make your program work.