Inf1B Object-Oriented Programming
Assignment 1 – loops, recursion and graphics

Hutchins

Out: January 19th, 2007   Due: February 2nd, 2007

Due Date: Friday, February 2nd, 2007

Materials: You will need to download `assignment1.zip` from the informatics 1 website:

Submission: Type the following command at the UNIX shell prompt to submit this assignment:

```
submit inf1 inf1b oop1 GraphMain.java
```

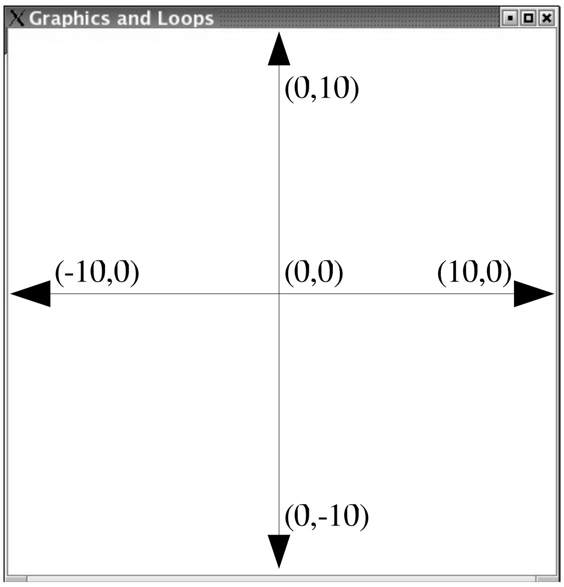
The fine print: If your program does not compile, 30% will be deducted from your mark. Late submissions will not be accepted. Collaboration with other students on this assignment is not permitted. If you have any questions on this assignment, please post them to the newsgroup `eduni.inf.course.inf1b` or ask a demonstrator in one of the drop-in lab sessions.

1 Introduction

This assignment is intended to improve your understanding of loops, variables, and recursion. To make things a bit more interesting, you'll be using loops, and recursion to draw pretty pictures on the screen, and even make some animated graphics!

1.1 Graphics

Java graphics commands are based on the same Cartesian coordinate system that that you used to draw graphs in your high-school algebra course. Each position in the window is represented by a point (x, y), where x is the horizontal coordinate, and y is the vertical coordinate. The point $(0, 0)$ is in the middle of the window, and the x and y axes span from -10 to 10. In other words, the bottom left corner of the window has coordinates $(-10, -10)$, while the top right corner has coordinates $(10, 10)$. All coordinates are given as real-valued numbers, which is type `double` in Java.



In order to actually draw anything on the screen, you need a *graphics object*. This object is created by the Java libraries, and passed to your methods as an argument. (Methods in Java are similar to functions in Haskell). In all the methods that you will be implementing, the graphics object is named g.

For this assignment, you may use the following commands:

- `g.drawPoint(` x, y `)` — draws a small point (a single pixel) on the screen.
- `g.drawLine(` x_1, y_1, x_2, y_2 `)` — draws a line from the point (x_1, y_1) to the point (x_2, y_2).
- `g.fillRect(` x_1, y_1, x_2, y_2 `)` — draws a filled rectangle, with its lower-left-hand corner at (x_1, y_1) and its upper-right-hand corner at (x_2, y_2).
- `g.fillCircle(` x, y, r `)` — draws a circle with a center at (x, y), and a radius of r.
- `g.setColor(` c `)` — sets the current drawing color. All subsequent lines, rectangles, and circles will be drawn in the new color. The color `c` can be one of the following: `Color.red`, `Color.yellow`, `Color.green`, `Color.blue`, `Color.black`, `Color.white`

1.2 Polar coordinates

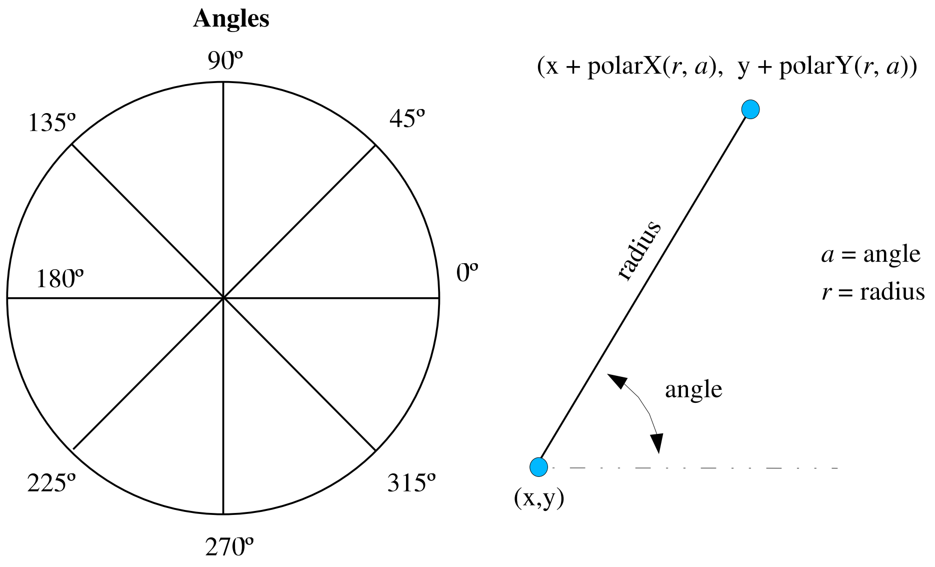
Updated For many pictures, such as spirals and polygons, a convenient way of calculating positions is to use *polar coordinates*. A polar coordinate is expressed in terms of a *radius* and an *angle*. The radius is the distance from the origin (i.e. point $(0, 0)$), and the angle is illustrated in the figure below. For example, a radius of 5 and an angle of 90 degrees specifies the Cartesian coordinate $(0, 5)$. The 90 degree angle points straight up the y axis, at a distance of 5 units.

You can use two functions to convert between polar and Cartesian coordinates:

- `polarX(` *radius*, *angle* `)` will return the Cartesian x-coordinate
- `polarY(` *radius*, *angle* `)` will return the Cartesian y-coordinate

For some of the problems in this assignment, you will need to find a point that is at a given angle and distance from some point (x, y), rather than from the origin.

You can do this by adding x and y to the result of `polarX` and `polarY`, as illustrated below.



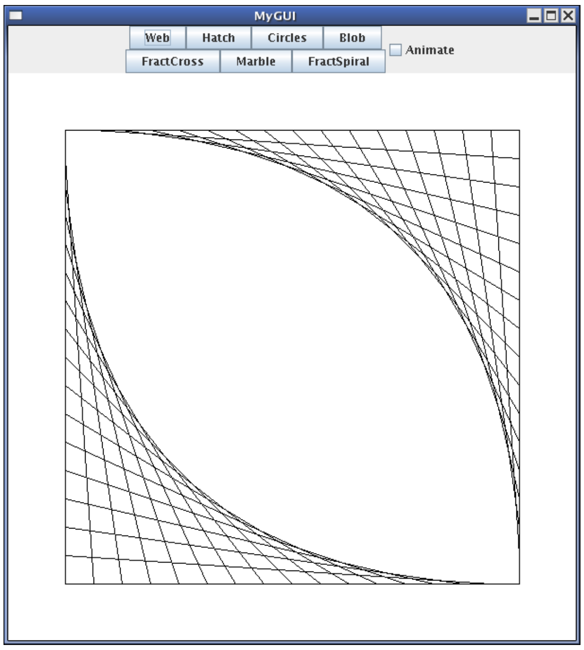
2 Assignment

To start the assignment, you will need to do the following:

- Download the example code from the inf1 website.
- Fire up eclipse, create a new project, and import the example code.
- Open the file `GraphMain.java`.

2.1 Problem 1: A tangled web. (10 points)

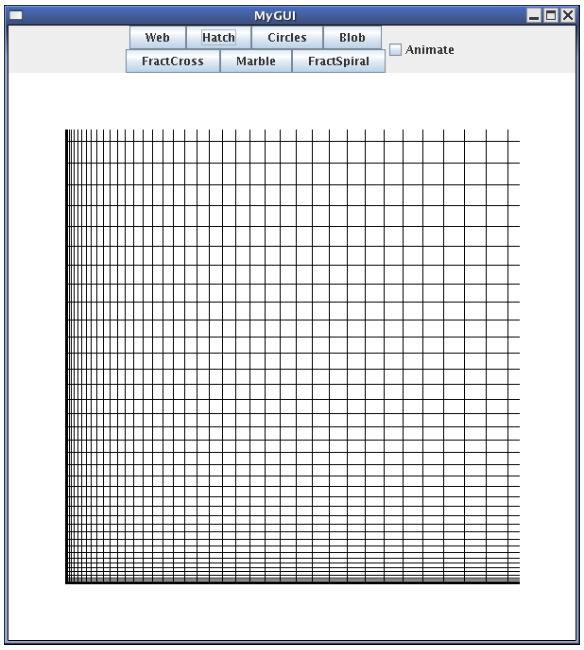
Find the method `drawWeb(GraphicPen g)` in `GraphMain.java`. Write an implementation that generates the following picture. The box shown below is 16 units on a side; coordinates range from -8 to 8 in increments of one. You should use one, and only one loop.



To see the web, run your program, and then click on the `<Web>` button.

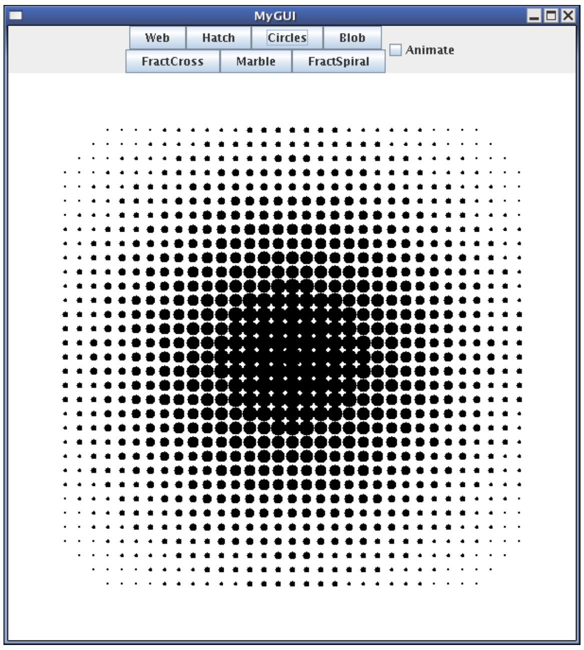
2.2 Problem 2: Hatch (10 points)

Find the method `drawHatch(g)`. Write an implementation that generates the following picture. The first two lines on the left (and bottom) are 0.02 units apart. The distance between each sucessive pair of lines is 0.02 units farther than the distance between the previous pair. You should use one and only one loop.



2.3 Problem 3: Circles (10 points)

Find the method `void drawCircles(g)`. Write an implementation that generates the following picture. (Hint – you will need to use two loops, one nested inside the other).

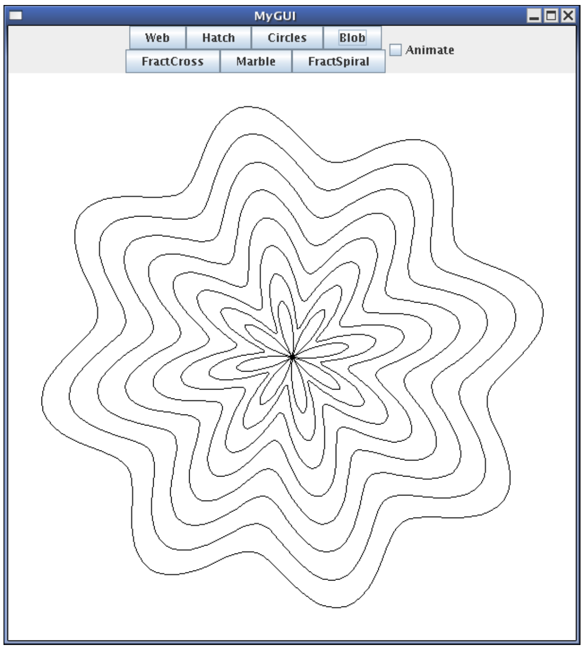


The small circles are placed 0.5 units apart, with coordinates that span from -8 to 8, inclusive. Each small circle at position (x, y) has a radius r given by the following equation:

```
r = 0.33 - Math.sqrt(x*x + y*y)/33
```

2.4 Problem 4: Blob. (15 points)

Find the method `drawBlob(g)`. Implement the body of this method to draw the following picture.



This picture has 8 wavy circles nested inside each other. The smallest one has a radius of 1, and the largest has a radius of 8.

A single wavy circle can be drawn with polar coordinates. The (x, y) position of each point on the circle is given by the following equations:

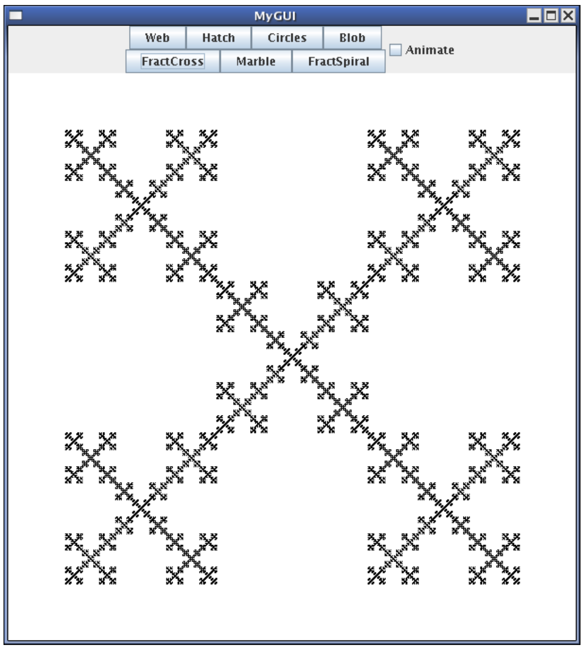
- Let `radius` be the radius of the wavy circle.
- Let `a` be an angle which ranges from 0 to 360, in increments of 2.
- `r = radius + blobiness * Math.sin(numBlobs*a*Math.PI/180)`.
- `x = polarX(r, a)`
- `y = polarY(r, a)`

In order to draw an unbroken circle, you will need to draw a line from each point on the circle to the previous point on the circle. You should use two variables, `oldx` and `oldy`, to store the location of the previous point.

2.5 Problem 5: Fractal cross. (10 points)

Now that you have mastered loops, it's time to move on to fractals. Fractals cannot be drawn with loops; instead, you will have to write recursive functions.

Find the method `fractCross(g, level, x, y, width)`. Implement it so that it draws the following picture.



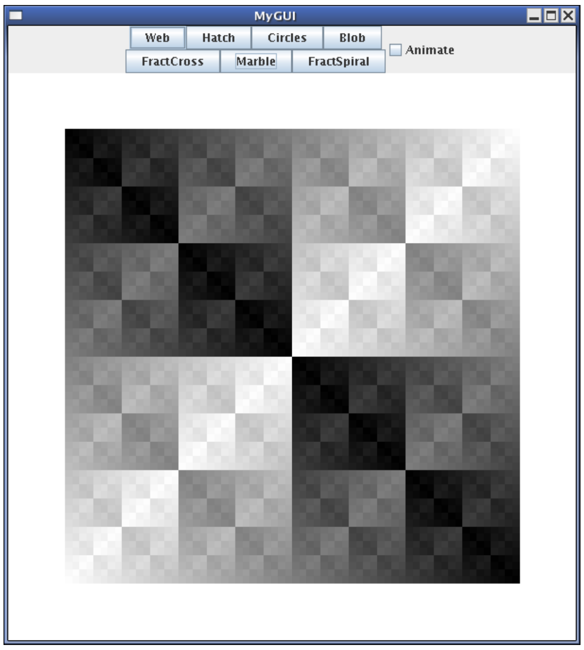
As you can see, this picture is a *fractal.* It is a Saint Andrew's Cross, but the cross is not made up of simple lines. It is made up of five smaller crosses, each of which is made of of 5 even smaller crosses, and so on.

The `fractCross` function has several arguments. The cross is centered at (x, y), and it has a width (and height) given by `width`. Each of the five smaller crosses has 1/3 the width of the original.

The `level` argument is used to control recursion. If `level` is zero, you should draw a simple cross with two lines. If `level` is greater than zero, then `fractCross` should call itself recursively with `level-1` as an argument. You will need to make five recursive calls, one for each of the five smaller crosses.

2.6 Problem 6: Marble. (15 points)

Find the method `marble(g, level, x, y, width, pcolor)`. Implement it so that it draws the following picture.



In the previous problem, the arguments to the function included a position (x, y), and a `width`. The `marble` function has an additional argument – color. It is not the shape of the fractal that is interesting, but its coloration. The square above is made up of four smaller squares, each of which is made up of four even smaller squares, and so on. Two of the squares have a darker shade than the original, and two have a lighter shade.

The `marble` method is first called with a gray color. This color can be *blended* with either white or black to make darker or lighter shades. You can blend `pcolor` with white using the following expression:

```
pcolor.blend(Color.white, 0.5)
```

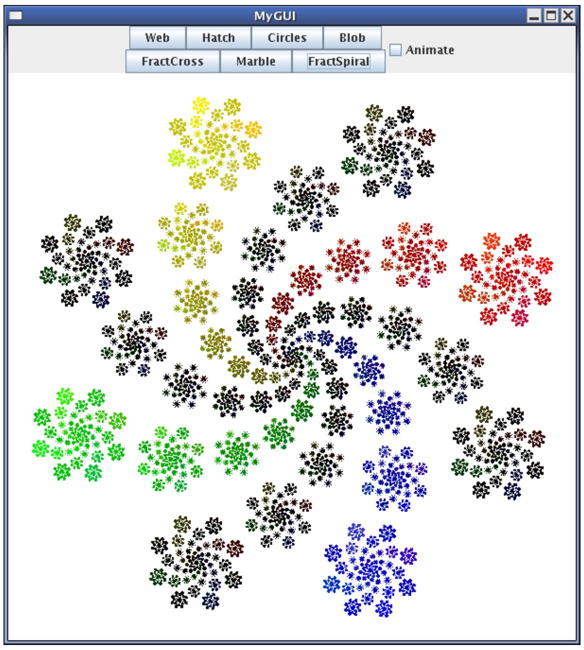
The number 0.5 controls the "transparency" of the blend. A value of 0 is transparent, while a value of 1 is opaque. Feel free to experiment with transparency, but be sure to use 0.5 when you submit.

When `level` reaches 0, you should use the `fillRect` command to draw a solid rectangle in the given color. You can't use `pcolor` directly, because it has special type that supports accurate color blending. You can convert `pcolor` to an ordinary color with the expression:

```
pcolor.getColor()
```

2.7 Problem 7: Fractal spiral. (20 points)

Find the method `fractSpiral(g, x, y, radius, pcolor)`. Implement the body of this method so that it draws the following fractal. The `(x,y)` arguments specify the center of the spiral, `radius` controls the size of the spiral, and `pcolor` is the color as before.



The outer edge of this fractal is a ring of smaller spirals. The number of small copies in the ring is given by `numSmaller`. Each copy is `smallSize` times smaller than the original, and should be positioned at a distance of `radius` from the center of the image. You will need to use polar coordinates to find the position of each small copy in the ring.

The center of the fractal consists of smaller and and smaller rings that spiral in towards the middle. Each ring is `radiusDec` times smaller than the one before it, and each one has been rotated by `twist` degrees.

You will need to use two loops and a recursive call in order to draw this fractal. Use recursion to draw each small spiral in the outer ring. Use one loop to draw the outer ring itself, and another loop to draw the inner rings spiraling in.

Unlike the previous two fractals, there is no `level` parameter which controls when to stop the recursion (and loops). Since all of the little spirals are different sizes, a fixed level would not given an even appearance. Instead, you should terminate the loop/recursion when the radius is smaller than the size of a single pixel — the smallest size that can be drawn. When the radius is smaller than `pixSize`, you should draw a single point at (x, y).

The small spirals in the outer ring should be different colors. The variable `armColors` is an array that holds the colors of each arm of the spiral. The color of the i^{th} arm is given by the expression `armColors[i]`, and you can blend colors use the same technique as before. The image above uses a transparency of 0.75 in the outer ring. It also blends the colors towards black as it spirals into the center, using a transparency of 0.15. Feel free to use your own colors if you want; you will receive full credit so long as you use color in some way.

2.8 Problem 8: Animations. (10 points)

Now that you know how to draw all of the images, it's time to animate them! An animation loop does the following three things over and over again.

- Draw the image.
- Wait a certain amount of time.
- Change the parameters of the image.

Because of the way in which the Java libraries work, it is not possible to call the draw routines directly. (You need a graphics context to do any drawing.) The `redraw()` command will tell the libraries to get a graphics context, and jump to the appropriate drawing method. The `delay(` n `)` command will then wait for n milliseconds.

You may have noticed that when you run your program, there is an `animate` box at the top of the window. When you check this box, it sets the `animate` variable to `true`, and then jumps to one of the animation routines. Your animation loop should run until either `animate` is false, or the user has selected something else to draw. The appropriate animation loops have already been provided.

Part (A): Find the `animateBlob()` method, and change the loop so that it increments `blobTime` by 2 on each iteration. This should make the blob jiggle.

Part (B): Find the `animateFractSpiral()` method, and change the loop so that it increments the `twist` variable by 0.5. This will make the fractal spiral around.

2.9 Bonus: (5 points)

For 5 points extra credit, choose one of the other drawing methods to animate. Animation loops are provided. You will probably need to add new variables to hold parameters of the image; look at how `blobTime` and `twist` are defined.