MAT 2170: Laboratory 8

Key Concepts

1. Developing algorithms
2. Writing and using methods

Exercises

1. (Page 172, Exercise 5, `SliderProgram`: `Bullseye`) Rewrite the original `Target` program to utilize the `createFilledCircle()` method. Get the number of circles to be drawn from the slider, with a range from 1 to 15. Be sure to include and use the constants OUTER_RADIUS and INNER_RADIUS, and follow all instructions in the exercise. There is no need to use a `pause` in this program — the finished image is the goal.

2. (Page 175, Exercise 11, Dialog Program: `PrimeSolver`) Create and test the `isPrime()` predicate method, then extend this exercise to ask the user for a range, `low` to `high`, and determine all the prime numbers in that range. These prime numbers are to be listed in a single dialog box with a message indicating what they are.

3. (From Denison University, Dialog Program: `MakePal`) An integer is a *palindrome* if its digits read the same left–to–right as right–to–left. For example, 23132 and 578875 are both palindromes. Note that single digit numbers are palindromes.

   Here is one way to produce a palindrome from any integer: if the integer is a palindrome, stop. Otherwise, reverse the digits in the integer and add 2, continuing until you get a palindrome (which you always will). For example suppose you start with 108. The sequence of numbers you'd get using the above process would be: 108, 803, 310, 15 (notice the leading zero is dropped when inverting), 53, 37, 75, 59, 97, 81, 20, 4. This took 11 steps.

   Write a program, `MakePal`, which asks for input from the user – a **series** of positive integers with two or more digits – and for each, displays the palindrome it produces (using the method described in the previous paragraph), and the number of steps it took to produce the palindrome. The expected display format is given in the example below. ***Beware of testing large numbers*** which aren't palindromes – they can take a lot of processing time.

   Each input should produce two lines of output (in a single dialog box). Use a sentinel to signal end of input. You must create and utilize at least two methods, `isPalindrome()` — to determine whether or not a number is a palindrome, and `reverseDigits()` — which reverses the digits in a number. The `isPalindrome()` method **should make use of** the `reverseDigits()` method, then compare the digits in the original with those in the reversed number. A sample Execution:

```
input:   output (in a dialog box):
108      The Palindrome generated for 108 is 4.
         This took 11 steps.

342      The Palindrome generated for 342 is 8.
         This took 13 steps.

1002     The Palindrome generated for 1002 is 111.
         This took 9 steps.

999      The Palindrome generated for 999 is 999.
         This took 0 steps.

-1       Thank you for using my program.  Bye.
```

4. (*Julia Sets*, `DualSliderProgram`: `JuliaSet`) You are to complete a program which produces a graphical display of a fractal, or Mandelbrot set. A fractal is a function that maps coordinates to values which may then be represented with various colors. The method which will do this mapping and coloration, `JuliaColor()`, is provided for you. The graphics window is to be divided into rows and columns in the same way we created a checker board and other tilings of the graphics window. In this case, however, the `JuliaColor()` method is used to determine the color for each block.

   In addition to completing the `run()` method, you must also complete the `BlockCorner()` and `ScreenToWorld()` methods. The **main idea** in this problem is to map each block's position in the graphics window to a corresponding point in the world region that we are representing. `BlockCorner()`, when given the row and column of a block, is to return the graphics window position of the block at the intersection of row and column. `ScreenToWorld()` is to calculate the world region coordinate associated with the current block's position in order to find the corresponding Julia color. `ScreenToWorld()` is passed a `GPoint` representing a point in the window, and it returns a `GPoint` representing the coordinates of the corresponding point in the world region.

   For each block in the grid, you:

   (a) find its upper left corner position (using method `BlockCorner()`)

   (b) find the corresponding point in the world region (using method `ScreenToWorld()`)

   (c) find the color this point (block) should get (using method `JuliaColor()`)

   (d) draw a block of the correct size, location, and color to represent the Julia map

   (The method `Norm()` is used by `JuliaColor()`, and the method `NextPoint()` is just the Julia map mentioned in lecture. There is no need to modify either of these methods.)

   Steps to Complete

   ☐ (a) Create a new project, `JuliaSet`, add the `acmLibrary.jar` to its libraries, create a new, empty java file as usual and copy the java code for the `JuliaSet` class skeleton into it from the course web site.

   ☐ (b) Locate the implementation of the `BlockCorner()` method (below `run()`). You are to complete this method so that given a row and column (as a `GPoint` parameter), `BlockCorner()` returns a `GPoint` indicating the position in the graphics window of the block at row and column. Add statements to the main program to test your code by filling the graphics window with cyan blocks with black borders. **Make use of the constants** which have been provided in the program skeleton. Now build and run the program, checking to see that everything works so far. **Correct any errors in this method before proceeding.**

   ☐ (c) You are already nearly finished. The last step will be to use the `JuliaColor()` function to determine the color of each `block` before it is drawn. To do this, you must complete the `ScreenToWorld()` method, used to scale a graphics window coordinate into a world region coordinate. The resulting `GPoint` should then be sent to the `JuliaColor()` method to determine the color of the current block.

Finishing Up

When you have completed the lab:

1. Publish the programs to your web site; submit an electronic copy of the lab

2. Print the programs, staple together, and hand in at the beginning of Lab 9.

Appendix

```
1   // add header comments here
2   import acm.graphics.*;
3   import java.awt.*;
4
5   public class JuliaSet extends DualSliderProgram {
6
7     public void init() {
8       setSize(700, 700);
9       super.init();
10      setRangeA(-75, 75);
11      setRangeB(0, 100);
12    }
13
14    public void run() {
15      // the a and b of F_a,b(x,y)
16      double a = getA() / 100.0;
17      double b = getB() / 100.0;
18      GPoint JuliaTerm = new GPoint(a, b);
19
20     // add code to draw Julia blocks here
21
22
23      }
24    }  // constants
25    public static final double SCREENSIZE = 700;    // Size of graphics window
26    public static final double WORLDSIZE = 4.0;     // Size of the "window" in the real world
27    public static final GPoint WORLDCENTER = new GPoint(0.0, 0.0);// Center of the world region
28    public static final int GRIDSIZE = 350;         // Number of blocks in screen grid
29    public static final double BLOCKSIZE = SCREENSIZE / GRIDSIZE; // size of squares
30    public static final int MAXCOLORS = 11;         // Number of colors used for the display
31    public static final int MAXITERATIONS = 40;     // Maximum number of iterations before a
32                                                    //   number is declared in the Julia set
33    public static final double THRESHOLD = 2.0;     // Distance from beyond which a point will
34                                                    //  not return
35
36    // calculate distance from point p to the origin
37    public double Norm(GPoint p) {
38      double x = p.getX();
39      double y = p.getY();
40      return Math.sqrt(x * x + y * y);
41    }
42
43    // next iteration of the Julia map
44    public GPoint NextPoint(GPoint p, GPoint JuliaTerm) {
45      double x = p.getX();
46      double y = p.getY();
47      return new GPoint(x * x - y * y + JuliaTerm.getX(),
48          2.0 * x * y + JuliaTerm.getY());
49    }
50
51    // returns the color of the point x, y, indicating how quickly it
52    // "escapes" under iterations of the Julia map
53    public Color JuliaColor(GPoint p, GPoint JuliaTerm) {
54      GPoint Z = new GPoint(p.getX(), p.getY());
55      int Iterations = 0;
56
```

```
57        while ((Norm(Z) < THRESHOLD) && (Iterations < MAXITERATIONS)) {
58          Z = NextPoint(Z, JuliaTerm);
59          Iterations++;
60        }
61
62        if (Iterations >= MAXITERATIONS) {
63          return Color.black;
64        } else {
65          switch (1 + Iterations % (MaxColors - 1)) {
66            case 1:
67              return Color.blue;
68            case 2:
69              return Color.cyan;
70            case 3:
71              return Color.green;
72            case 4:
73              return Color.red;
74            case 5:
75              return Color.yellow;
76            case 6:
77              return Color.orange;
78            case 7:
79              return Color.magenta;
80            case 8:
81              return Color.pink;
82            case 9:
83              return Color.white;
84            case 10:
85              return Color.dark_gray;
86            default:
87              return Color.green;
88          }
89        }
90      }
91
92      // the position of the world point corresponding to the screen point
93      public GPoint ScreenToWorld(GPoint p) {
94        // replace this code
95        return new GPoint(0.0, 0.0);
96      }
97
98      // position of block at row, col
99      public GPoint BlockCorner(int row, int col) {
100       // replace this code
101       return new GPoint(0.0, 0.0);
102     }
103   }
```