CS 100: Programming Assignment P1

Due: Thursday, February 4. (Either in lecture or in Carpenter by 4pm.)

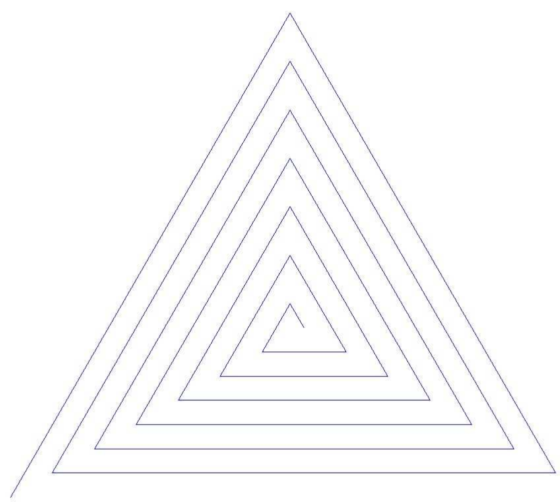
You may work in pairs. Uphold academic integrity. Follow the course rules for the submission of assignments.

Introduction

This assignment has three parts each of which involves playing with a given Java program. Our goal is for you to get familiar with the CS 100 Code Warrior environment. To complete the assignment you will have to understand the handout *Guide to Code Warrior Java for CS100 and CS 211*. That document will be referred to as *The Guide* below.

Part A. Logarithmic Spirals (3 points)

Here is an example of a discrete logarithmic spiral:



It has twenty "legs" which we index from 1 to 20 starting from the inside. The *k*-th leg has length *k*d* where *d* is the length of the first leg. Each leg makes a fixed angle with its predecessor. Indeed, if you were to walk along the spiral you would turn left 120 degrees at the end of each leg. The Java program P1A (listing attached) draws a logarithmic spiral with 1000 legs, turn angle 137, and initial leg length d = .4. So that you can work with this program follow these steps:

- Get a copy of P1A from the CS 100 website. Do this by (1) following the links to Program Assignment P1, (2) highlighting the program using the mouse, and (3) clicking the mouse.
- Set up a project using the CUCS Java Graphics Application stationary. (See Section 2.1 of *The Guide*.)

- Create a file MyP1A.java (See Section 2.2.1 of *The Guide*.) and paste into it the code obtained from the website.
- Add MyP1A.java to the project. (See Section 2.2.2 of *The Guide*.)
- Remove CUCSGraphicsApplication.java from the project. (See Section 2.2.5 of *The Guide*.)
- Change the "main class" to P1A. (See Section 2.2.4 of *The Guide*.)
- Run the program. (See Section 1.3 of *The Guide*.)

You should see a spiral displayed in the graphics window.

Read over P1A to get a general impression. You are not expected to understand why everything works. That will take a couple of lectures. Deliberately introduce errors via misspellings and whole line deletions. The latter is easily accomplished by placing a double slash `//` at the beginning of the line. Run the sabotaged code to see what happens. Controlled experimentation in this way will teach you something about how CodeWarrior responds to errors.

Assuming that you have restored the program to its original form, locate the code fragment where the "design parameters'' of the spiral are established:

```
final int n = 1000;
final int turn = 137;
final double d = .4;
```

Experiment with different values. Don't forget to quit the Java environment before you rerun the code. (See Section 1.3 of *The Guide*.)

By now you have noticed that the program produces a 3-color spiral. Indeed the *k*-th leg is red, blue, or green according to whether the remainder of *k* divided by 3 is 0, 1, or 2. Play with the choice of colors just as you played with the numerically-valued design parameters `turn`, `n`, and `d`. The Java list of preset colors include

```
black       blue        cyan        gray
darkgray    lightgray   green       magenta
orange      pink        red         white
yellow
```

For example, you can change

```
g.setColor(Color.red)
```

to

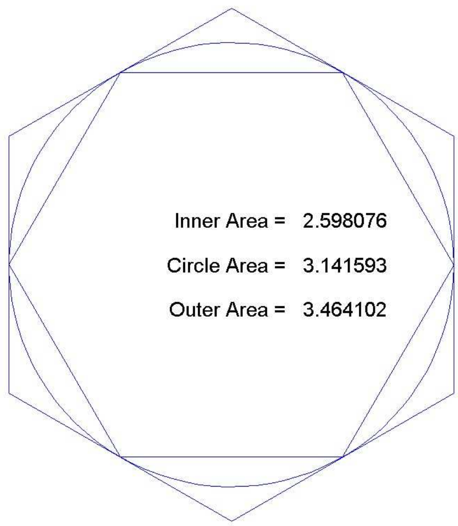
```
g.setColor(Color.magenta)
```

if you prefer magenta to red. If you can figure out how experiment with the *number* of colors that are involved in your spiral. (For example, if *k* divided by 5 has remainder 0,1,2,3, or 4 then set the color to cyan, orange, green, blue and red respectively.)

Submit a copy of your "best" spiral and a listing of your modified program MyP1A.java that produced it. (See Section 1.4 of *The Guide* on how to do this.) You will not be graded on the aesthetic value of your spiral. Indeed, what looks good on the screen may not be so pleasing when it is printed out because of the grayscale conversions of the colors and the typical loss of detail when you go to the printed page.

Part B. Estimating π Via Polygon Approximation (3 points)

Here is a picture of the unit circle, an inscribed regular hexagon, and a circumscribed regular hexagon:



Clearly, the two polygon areas straddle the value of π which is the area of the circle. There is nothing special about hexagons. Let A(n) be the area of the inscribed regular *n*-gon and let B(n) be the area of the circumscribed regular *n*-gon. The program P1B (listing attached) sheds light on how well these quantities approximate π as *n* gets very large. So that you can work with this program follow these steps:

- Get a copy of P1B from the CS 100 website. Do this by (1) following the links to Program Assignment P1, (2) highlighting the program using the mouse, and (3) clicking the mouse.
- Set up a project using the CUCS Java Application stationary. (See Section 2.1 of *The Guide*.)
- Create a file MyP1B.java (See Section 2.2.1 of *The Guide*.) and paste into it the code obtained from the website.
- Add MyP1B.java to the project. (See Section 2.2.2 of *The Guide*.)
- Remove CUCSApplication.java from the project. (See Section 2.2.5 of *The Guide*.)
- Change the "main class" to P1B. (See Section 2.2.4 of *The Guide*.)
- Run the program. (See Section 1.3 of *The Guide*.)

You should see a rather sloppy table displayed in the console (text) window. The table is sloppy because the `System.out.println` statement that displays the values of `n`, `InnerA`, `OuterA`, and `Error` doesn't do a good enough job of fomatting. To solve this problem, remove this statement and replace it with these four statements:

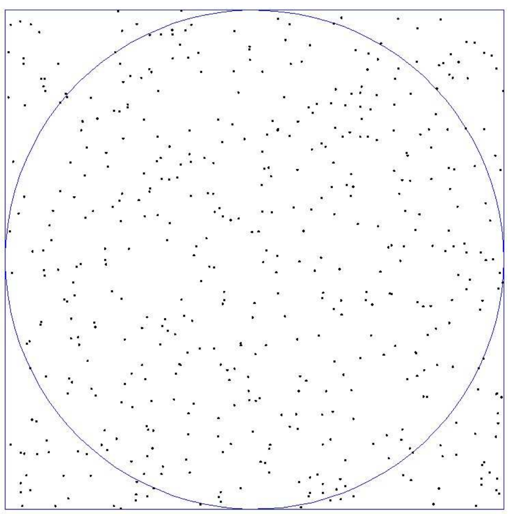
```
Format.print(System.out,"???",n);
Format.print(System.out,"???",InnerA);
Format.print(System.out,"???",OuterA);
Format.println(System.out,"???",Error);
```

Replace the question marks with appropriate format strings so that (1) `n` is displayed as an integer, (2) `InnerA` and `OuterA` are displayed as decimals with nine places to the right of the decimal point, and (3) `error` is displayed in scientific notation with a 3-digit mantissa. The columns in the final table should be right-justified and sufficiently spaced apart. (The data in a column is right-justified if the entries are lined up along their right edge.) The column headings should be appropriately centered over the data. In short, make the output table look nice.

Submit a copy of the output and listing of MyP1B.java used to produce it. (See Section 1.4 of *The Guide* on how to do this.)

Part C. Estimating π Via Monte Carlo (4 points)

Consider the following "dart board":



Suppose we throw a large number of darts and that they land randomly on the square. Let f be the fraction that land inside the circle and note that f should approximate the quotient of the circle's area to the square's area, i.e., $f \cong \pi/4$. Thus, since f = "hits"/total throws is known we can estimate π from the formula $\pi \cong 4*f$. The program P1C (listing attached) simulates this "dart game" and produces an estimate of π. So that you can work with this program follow these steps:

- Get a copy of P1C from the CS 100 website. Do this by (1) following the links to Program Assignment P1, (2) highlighting the program using the mouse, and (3) clicking the mouse.
- Open the project you created for Part A (See Section 1.1.1 of *The Guide*.)
- Create a file MyP1C.java (See Section 2.2.1 of *The Guide*.) and paste into it the code obtained from the website.
- Add MyP1C.java to the project. (See Section 2.2.2 of *The Guide*.)
- Remove MyP1A.java from the project. (See Section 2.2.5 of *The Guide*.)
- Change the "main class" to P1C. (See Section 2.2.4 of *The Guide*.)
- Run the program. (See Section 1.3 of *The Guide*.)

You should see the dart board displayed in the graphics window and the π-estimate in the console window.

`P1C` implements a single simulation and it displays both the darts throws and the estimate of π. Modify the program so that it performs two simulations. The first should involve 100 dart throws and it should just display the darts. (No tabulation of "hits" is required and it should not report a π-estimate.) The second simulation should involve 100,000 throws and it should just report the π estimate. (The dart throws should not be displayed.)

The easy way to do this is to (a) copy the code in between dotted comment line "A" and dotted comment line "B" and (b) paste it in between dotted comment line "B" and dotted comment line "C". Then edit the two simulation fragments so that they perform as required. There is no need to alter any of the statements that precede dotted comment line "A".

If the printer you are using is "too granular" to display the dart throws, then change the `g.drawOval` statement so that larger dots are drawn, e.g.,. `g drawOval((int) x,(int) y,3,3).`

Also, feel free to use more than 100,000 throws in the second simulation. Indeed, you might want to discover how the π-estimate improves with increasing *n*.

Submit a listing of your MyP1C.java and a copy of the output it produces, i.e., a copy of the console (text) window showing the π estimate and a copy of the graph window showing the 100 dart throws. (See Section 1.4 of *The Guide* on how to do this.)

```
// P1A
// Draws a logarithmic spiral with variable color..

import java.io.*;
import java.awt.*;

public class P1A {
   public static void main(String args[]) throws IOException {
      Drawing d = new Drawing();
         d.resize(600,500);
         d.move(0,75);
         d.setTitle("P1A: Logarithmic Spiral");
         d.show();
         }
   }

public class Drawing extends Frame {

   final int hc = 300;
   final int vc = 250;    // (hc,vc) = center of the spiral


   final int n = 1000;    // The total number of "legs" in the spiral.
   final int turn = 137;  // Turn factor. Each leg turns "left" this amount in degrees
   final double d = .4;   // The length of the kth leg is k*d

   public void paint(Graphics g) {

      int h, v;             // The horizontal and vertical coordinates of the current
                            //   vertex.
      int h_next, v_next;   // The horizontal and vertical coordinates of the next vertex.
      int theta;            // The "angle" of the current leg.
      int k;                // The index of the current leg.
      double L;             // The length of the current leg.

      h = hc;
      v = vc;
      for (k=1;k<=n;k++) {
          // Determine color of the kth leg.
          if (k%3==0)
             g.setColor(Color.blue);
          else if (k%3==1)
             g.setColor(Color.red);
          else
             g.setColor(Color.green);
          // Determine endpoints of the kth leg and draw.
          theta = k*turn;
          L = k*d;
          h_next = (int) Math.round(h + L*Math.cos(theta*Math.PI/180));
          v_next = (int) Math.round(v + L*Math.sin(theta*Math.PI/180));
          g.drawLine(h,v,h_next,v_next);
          h = h_next;
          v = v_next;
          }
       }
    }
```

```java
// P1B
// Estimates pi as a limit of polygon areas.

import java.io.*;

public class P1B {
   public static void main( String args[]) {

      int n;                   // The number of sides in the approximating n-gons.
      double OuterA, InnerA;   // The areas of the circumscibed and inscribed n-gons.
      double Error;            // OuterA - InnerA
      double c, s;             // The cosine and sine of pi/n.

      n = 4;

      OuterA = 4;
      InnerA = 2;
      c = 1.0/Math.sqrt(2.0);
      System.out.println("\n    n             A(n)                B(n)         B(n)-A(n)");
      System.out.println("------------------------------------------------------------");
      while (OuterA - InnerA > 0.00000001) {
         n = 2*n;
         s = Math.sqrt((1.0 - c) / 2.0);
         c = Math.sqrt((1.0 + c) / 2.0);
         InnerA = n*s*c;
         OuterA = n*s/c;
         Error = OuterA - InnerA;
         System.out.println(n + "    " +  InnerA + "    " + OuterA + "    " + Error);
     }
     System.out.println();
   }
}
```

```
// P1C
// Estimates pi via Monte Carlo.

import java.io.*;
import java.awt.*;

public class P1C {
   public static void main(String args[])  {
      Drawing d = new Drawing();
          d.resize(600,500);
          d.move(0,75);
          d.setTitle("P1C: The Dart Board");
          d.show();
          }
   }

public class Drawing extends Frame {

   final int hc = 300;
   final int vc = 250;         // (hc,vc) = center of circle
   final int r  = 200;         // the radius

   public void paint(Graphics g) {

      int Left = hc-r;         // Left edge of square
      int Top  = vc-r;         // Top edge of square

      long k;                  // Dart throw index
      long hits;               // Number of darts that land in circle.
      long n;                  // Total number of darts thrown.

      double x,y;              // (x,y) = coordinates of current dart throw
      double dist;             // Distance of current dart throw to center.
      double MyPi;             // The pi-estimate.

      // Draw the target.

      g.drawOval(hc-r,vc-r,2*r,2*r);
      g.drawRect(hc-r,vc-r,2*r,2*r);

      // --------------- Comment Line A ----------------------

      hits = 0;
      n = 1000;
      for (k=1;k<=n;k++){
         // Generate the next dart throw and plot.
         x = Left + 2*r*Math.random();
         y = Top  + 2*r*Math.random();
         g.drawOval((int)x,(int)y,1,1);
         // Check to see if the throw lands inside the circle and tabulate.
         dist = Math.sqrt((x-hc)*(x-hc) + (y-vc)*(y-vc));
         if (dist<=r)
            hits++;
      }
      MyPi = ((double) hits/n)*4;
      Format.println(System.out,"Number of dart throws = %6d\n",n);
      Format.println(System.out,"Appx Pi = %20.15f",MyPi);
      Format.println(System.out,"True Pi = %20.15f",Math.PI);
      System.out.println();

      // --------------- Comment Line B ----------------------


      // --------------- Comment Line C ----------------------


    }
 }
```