CS 145 Half-Homework 1
Problems Three

1 Overview

Your objective in this homework is to steep yourself in the world of mathematical calculation using a programming language. Unlike your calculator math, however, math in code is a little different:

1. It makes considerable use of variables, whose names tend to be longer and more meaningful.
2. Numbers are typed. Some number types allow fractional components, others are subsets of integers. Other types of values exist too; most aren't even numbers.
3. Some operators, like the remainder operator (`%`), aren't common in math. Other operations don't have operators; instead they are tucked away inside methods of the `Math` class. This is even true of math with functions like sine and cosine.
4. In code, we focus more on the *process* of producing an answer.

You will encounter these differences as you solve the following three problems of calculation.

2 Requirements

Complete the three classes described below. Place all classes in package `hw1`. Note that some requirements are about your program's output, while others are about its behavior—the code you write to achieve the output.

2.1 Paperclips

In 2000, Jeanine Van der Meiren set out to break the world record for the longest paperclip chain ever assembled. After 24 hours, Jeanine's chain contained 22025 paperclips. The record was hers. But how safe was it? Could you do better? 22025 sounds like a big number...

In this exercise, you will investigate writing code to convert Jeanine's total into a form that will help you see how you might fare against her—as seconds per paperclip instead of paperclips per 24 hours.

1. Write your code in a main method in a class named `Paperclips`.
2. Create declassignments for Jeanine's record: `nHours` and `nClips`. Also create declassignments for time conversion: `minutesPerHour` and `secondsPerMinute`. All are `int`s. Use literals to assign these variables their correct values.
3. Create declassignments for derived data: `hoursPerClip`, `minutesPerClip`, and `secondsPerClip`. In assigning values to these variables, use expressions comprised only of operators and other variables. No literals may appear.
4. Print `hoursPerClip`, `minutesPerClip`, and `secondsPerClip` on the first three lines of output, using this format:

```
Hours per clip: #
Minutes per clip: #
Seconds per clip: #
```

Replace the #s with the appropriate variable references. Match exactly the spelling, case, and whitespace.

5. Find a handful of paperclips. Practice chaining them together a few times.
6. Time yourself adding one paperclip to a chain. How many seconds did it take? Can you compete with Jeanine?
7. Now go for a longer stretch. Give yourself one minute to build a chain. How long is it?
8. Assuming you can sustain the pace you kept for that one minute, calculate how long you'd expect your chain to be after 24 hours. Print the result as an `int` on a line after Jeanine's rates from above in this format:

```
My total: #
```

2.2 Lissajous

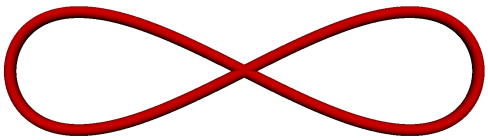
There are many ways to represent mathematically-defined structures:

1. With *explicit* equations. Functions, which map values of x to y, are often expressed as explicit equations. For example, one can turn kilograms into pounds with the explicit equation $y = 2.2x$.
2. With *implicit* equations. Paths that have multiple y values for a single x value are not functions and must be expressed in a different manner. Implicit equations allow this but relate x and y in a less direct manner. For example, all the points on a circle satisfy the implicit equation $radius^2 = x^2 + y^2$.
3. With *parametric* equations. These combine the directness of explicit equations with the ability to model meandering, non-function paths of implicit equations. Parametric equations are defined on a different parameter, which we often call t. In two dimensions, we need two functions to define a structure: one to yield an x value and another to yield the corresponding y value. For example, a circle can be defined parametrically with the following system, with t in $[0, 2\pi]$:

$$\begin{aligned} x(t) &= radius \cdot \cos t \\ y(t) &= radius \cdot \sin t \end{aligned}$$

Parametric equations are a great boon to computer scientists, who often find themselves needing to generate shapes programmatically in code. We can step along the range of t, evaluate the equations, and draw line segments between the resulting coordinate pairs. The figure 8 in figure 1 was generated using Lissajous equations.

Your task in this exercise is to generate and print coordinate pairs for several locations on this figure 8. In particular, you are expected to:



$$\begin{aligned} x(t) &= 6\sin t \\ y(t) &= \sin 3t \end{aligned}$$

Figure 1: A figure 8 and its generating parametric equations.

1. Write your code in a main method in a class named `Lissajous`.
2. Evaluate the given parametric equations at the following values of t, given in degrees: [0, 40, 80, 120, 160, 200, 240, 280, 320, 360]. Let the computer perform all arithmetic. Write expressions based on the given equations.
3. Print the 10 coordinate pairs to the console in the following form:

   ```
   (x1,y1),
   (x2,y2),
   ...
   (x10,y10)
   ```

   The x and y values will typically have fractional components. Consider them `double`s. Allow `System.out` to decide how many decimal digits to print. Copy and paste the output into Desmos (`http://www.desmos.com`) to visually test your output.

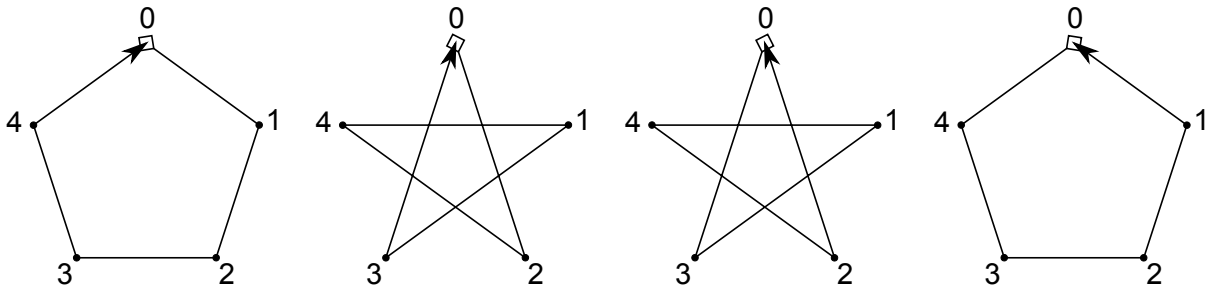
2.3 Star Winding

Imagine you've got five points evenly distributed across the perimeter of a circle. Let's call the points *nodes*. One of the nodes, which we'll call node 0, is at the top of the circle. The rest are numbered clockwise from node 0. You decide to trace out a path that starts at node 0, jumps from it to the next, and keeps on doing so until we return to node 0. The result is a conventional pentagon, as shown in figure 2a. Then you wonder what happens if, after starting at node 0, you skip node 1 and go directly to node 2. From node 2, you skip to node 4. From node 4, you skip ahead two nodes again to node 1, and so on. The result is five-pointed star, as shown in figure 2b.

You do this again, but now jump three nodes with every bend, visiting nodes in the order 0, 3, 1, 4, 2, until you arrive back at 0. Again, you have traced a star, as shown in figure 2c. Jumping by four visits nodes 0, 4, 3, 2, 1, producing the pentagon shown in figure 2d.

How do the paths behave for larger numbers of nodes? In particular, what do the paths look like for a 12-node figure, a dodecagon? For this exercise, your task is to write code that visits the nodes of a dodecagon to form a 12-pointed star. Each node must be visited exactly once, which eliminates some jumping patterns. Follow these steps:

1. Write your code in a main method in a class named `StarWinding`.
2. Print the nodes in an order that produces a 12-pointed star. You will need to experiment and reason about how many nodes you should jump, since not just any jump value produces a star or ensures that all nodes are visited. On the pentagon, for example, 1-node and 4-node jumps do not produce a star. Nothing beats pencil and paper.



(a) 1-node jump (b) 2-node jump (c) 3-node jump (d) 4-node jump

Figure 2: The possible clockwise orderings of nodes of a pentagon, all starting with node 0 and jumping to successor nodes until node 0 is reached again. Note that 1-node and 4-node jumps do not yield a self-intersecting path and therefore do not produce stars.

3. Print the nodes on just one line, separated by spaces. For the pentagon, either of the following would be acceptable output.

   ```
   0 2 4 1 3
   ```

   ```
   0 3 1 4 2
   ```

   Trailing whitespace is allowed.

4. Let the computer do the arithmetic. I encourage you to first find an order without the help of the computer, but your code should recreate your solution mechanically, using arithmetic to calculate each next node in the ordering. In fact, your algorithm should look something like the following:

   ```
   initialize any variables
   print current node
   jump
   print current node
   jump
   print current node
   jump
   // repeat 12 times, all told
   ```

   Jumping may be accomplished with a short mathematical expression, using the operators we've discussed in lecture. Note what happens when we jump past the last node: we wind back around. What operator exhibits that behavior?

3 Submission

To submit your work for grading:

1. Put the SpecChecker for this homework in your Build Path. Run the SpecChecker as a Java Application and fix problems until all tests pass.

2. Commit and push your work to your repository. Verify that your solution is on Bitbucket.

A passing SpecChecker does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The SpecChecker checks some of them, but not all.

- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, CS 145 moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

- Your code must be submitted correctly and on time. Most excuses devolve into, "I started too late." The fix for this problem is not an extension.