CS 1 Half-Homework 1
Main Train

1 Overview

Your objective in this homework is to acquaint yourself with the world of mathematical calculation using a programming language. Math in code is a little different than the calculator math you are used to in the following ways:

1. Programmers make considerable use of variables, whose names tend to be longer and more meaningful.
2. Numbers are typed. Some number types (like `double`) allow fractional components, others are subsets of integers (like `int`).
3. Other types of data exist too (like `String`s).
4. Some operators, like the remainder operator (`%`), aren't as commonly used in the math you've learned in school.
5. Mathematical functions like `sin` and `cos` are available, but they are defined inside the class `Math` and must be invoked through it (for example, `Math.sin(0)`).
6. In code, we focus more on the step-by-step *process* of producing an answer and less on stating mathematical truths.

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 the code you write to achieve the output.

2.1 LemonAid

Class `LemonAid` helps a user scale a lemonade recipe. Its `main` method asks the user for a parts-based recipe (the relative proportions of lemon juice, sugar, and water) and the desired number of cups of lemonade. It prints out the amount of lemon juice, sugar, and water needed to produce the desired amount of lemonade. Match the output shown in this example *exactly*—even the whitespace:

```
How␣many␣parts␣lemon␣juice?␣1
How␣many␣parts␣sugar?␣2
How␣many␣parts␣water?␣3
How␣many␣cups␣of␣lemonade?␣12
Amounts␣(in␣cups):¶
␣␣Lemon␣juice:␣2.0¶
␣␣Sugar:␣4.0¶
␣␣Water:␣6.0¶
```

Where you see a ¶, insert a linebreak by using `println` (or `printf` and `%n`).[1] Where you see a ␣, insert a space character.

In this example, the proportions tell us that there are 6 parts total, and that lemon juice makes up 1 part, or $\frac{1}{6}$ of the mixture. Applied to the 12 total cups, we need 2 cups of lemon juice.

The input and output, of course, will change based on what the user enters. Read from `System.in` and write to `System.out`. The last three lines are indented two spaces. It is not your program's fault if the lemonade tastes bad. If users enter unreasonable values (like negative sugar or zero water), that is their prerogative.

Assume the user enters four integers. Read them with `Scanner`'s `nextInt` method, and store them in variables of type `int`. Calculate the total number of parts and store it also in an `int`. Next, calculate the three proportions and store them in `double` variables. Watch your arithmetic! An `int` divided by an `int` produces an `int`. Finally, apply these proportions to the total number of parts to calculate the overall amounts of each ingredient. Follow this order of operations to make sure we arrive at the exact same output.

Do nothing special to format the `double` output. Just print your amounts directly with `System.out`'s `println` method.

2.2 UshSure

Class `UshSure` helps ushers employed by chain of theaters seat guests. Tickets were printed by the central office, but since every theater in the chain is a little bit different, the tickets do not show seat numbers. Instead, only a more general serial number appears on them: 0, 1, 2, 3, 4, and so on. The seats themselves are marked A0, A1, A2, ..., B0, B1, B2, and so on. The seats are arranged in a rectangle. If the rows are 5 seats wide, for example, the arrangement is as follows:

	Column				
Row	*0*	*1*	*2*	*3*	*4*
A	0	1	2	3	4
B	5	6	7	8	9
C	10	11	12	13	14
D	15	16	17	18	19
E	20	21	22	23	24
F	25	26	27	28	29

The non-italicized numbers are the ticket numbers. The italicized numbers mark the seat columns and the letters mark the seat rows.

A ticket number is mapped to a row number by calculating how many *full* rows precede the given seat. The seat for ticket 23, for example, is preceded by 4 rows. Mathematically, one can determine this number by asking how many 5s go into 23. We can go 5, 10, 15, 20, so the answer is 4. Whatever is left over is the seat's column number, which in this case is 3. To turn a row number into a row letter for printing, add it to '`A`'. By the rules of Java, the resulting value is an `int`, but we can cast it back to a `char`.

The `main` method of this class asks an usher for the ticket number and the number of seats in a row in the usher's theater. It converts the one-dimensional serial number into a two-dimensional seat number and prints it. Match the output shown in this example *exactly*—even the whitespace:

[1] Some folks want to use `\n` or `\r\n` to insert a linebreak, but neither is standard across all major operating systems. Make your code more portable by not using them.

```
Ticket␣number?␣ 23
Seats␣in␣a␣row?␣ 5
Seat:␣E3¶
```

The input and output, of course, will change based on what the user enters. Read from `System.in` and write to `System.out`. Assume the user enters valid integers.

2.3 Shuriken

If you received a mathematics education similar to mine, you probably talked about functions almost every class period with Mr. Albertson. (The only deviation from this routine was when Kirby D. swallowed the goldfish from Mr. Albertson's fishtank.) These functions were always named f or g. We represented these functions with *explicit* equations that clearly described the algorithm for computing y from parameter x. For example, we modeled a parabola with the explicit equation $y = x^2$.

Discussed far less were *parametric* equations, which compute both x and y based on some other parameter, often called t. These equations could express many more structures, including paths that circled back around to previous x values—something that functions cannot do. Circles, for instance, cannot be expressed with an explicit equation. However, a circle with radius r can be defined parametrically with t in $[0, 2\pi]$:

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

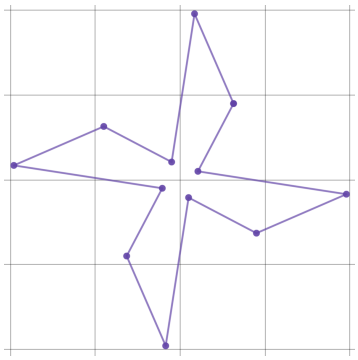
What's nice about parametric equations is that they lend themselves to be "walked" in code quite naturally. The following algorithm steps through the path defined by the parametric equations:

```
t = first value of t
x = compute x based on t
y = compute y based on t
print x, y

t = next value of t
x = compute x based on t
y = compute y based on t
print x, y
...
```

Note that we must know the interval on which t is defined to be able to start and stop this code. We also won't be able to visit every possible value of t in the given interval, as there are an infinite number of them. How often we sample the interval can lead to various results.

The `main` method of class `Shuriken` walks the parametric equations shown in Figure 1. Let t span the interval $[0, 360]$, stepping every 30 degrees. Note that while t is defined as a number of degrees, `Math.sin` and `Math.cos` expect a number of radians. Variable `spin` is a `double` value from the user that rotates the shuriken. Match the output shown in this example with a spin of 45 degrees *exactly*—even the whitespace:



$$\begin{aligned} x(t) &= (1.1 + \sin 4t) \times \cos(t + \text{spin}) \\ y(t) &= (1.1 + \sin 4t) \times \sin(t + \text{spin}) \end{aligned}$$

Figure 1: A shuriken (spun 55 degrees) and its generating parametric equations.

```
Spin? 45
0.78,0.78¶
0.51,1.90¶
-0.06,0.23¶
-0.78,0.78¶
-1.90,0.51¶
-0.23,-0.06¶
-0.78,-0.78¶
-0.51,-1.90¶
0.06,-0.23¶
0.78,-0.78¶
1.90,-0.51¶
0.23,0.06¶
0.78,0.78¶
```

Use `System.out`'s `printf` method to control the number of digits after the decimal point in the output. For example, to print the line `pie: 3.1415 2.7`, one can write:

```
System.out.printf("pie: %.4f %.1f%n", Math.PI, Math.E);
```

The `printf` method expects as its first parameter a format string that contains a mixture of literal text and special placeholders for data that begin with a `%` character. The sequence `%.4f` tells `printf` to insert the next *floating point* number—that is, a number with a fractional component—with four digits after the decimal.

To test your result, visit Desmos. Copy only the numeric lines and paste them into a cell in the left panel. Only the vertices of the walked path will appear. To connect them with lines, click on the gear icon. Above the y column is a colored circle. Click on it and change the style accordingly.

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.