

Assignment #2: Simple Java Programs

Due: 11AM PST on Monday, April 23rd

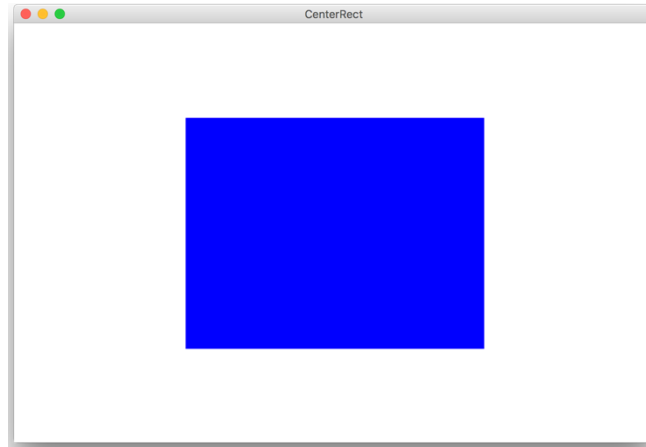
This assignment should be done individually (not in pairs)

Portions of this handouts by Eric Roberts

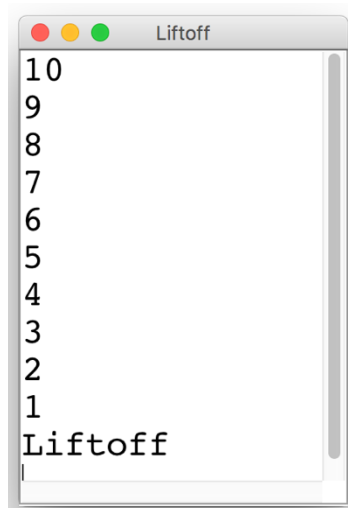
Your job in this assignment is to write programs to solve each of these nine problems. You should start by downloading the starter project for Assignment #2 from the CS106A assignment page (go to the CS106A web site and click the **Assignments** link). The starter project will provide java files for you to write your programs in.

Warmups

1. As warmup, write a **GraphicsProgram** that draws a blue, filled rectangle with width equal to 350 pixels and height equal to 270 in the center of the screen:



2. As warmup, write a **ConsoleProgram** that prints out the calls for a spaceship that is about to launch. Write the numbers 10 down to 1 and then write liftoff. You must use a loop:



Console Challenges

3. In high-school geometry, you learned the Pythagorean theorem for the relationship of the lengths of the three sides of a right triangle:

$$a^2 + b^2 = c^2$$

which can alternatively be written as:

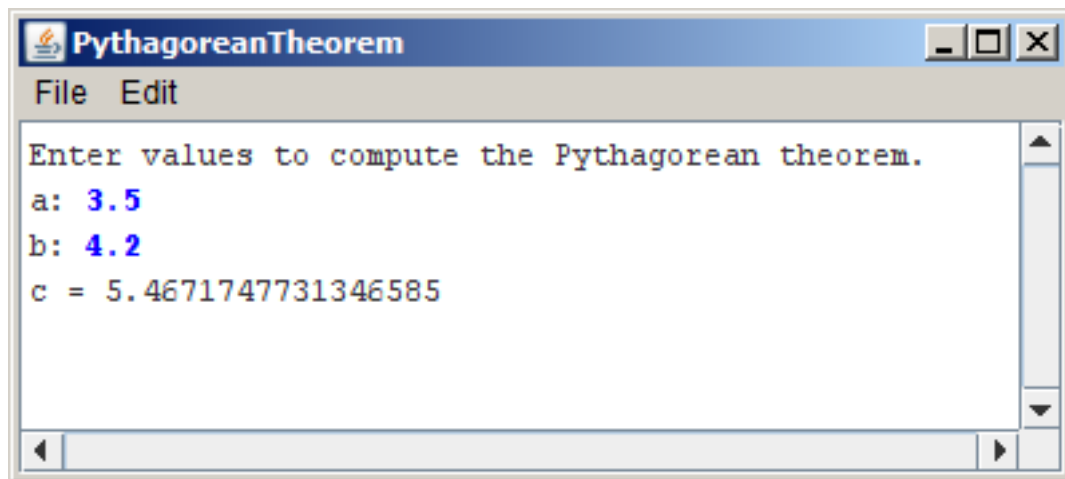
$$c = \sqrt{a^2 + b^2}$$

Most of this expression contains simple operators covered in Chapter 3. The one piece that's missing is taking square roots, which you can do by calling the standard function **Math.sqrt**. For example, the statement

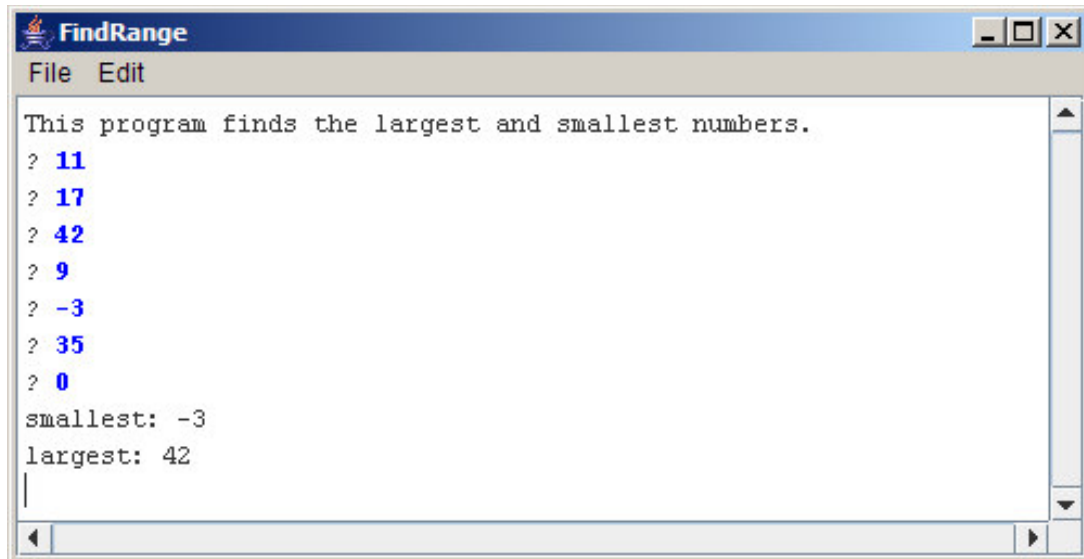
```
double y = Math.sqrt(x);
```

sets **y** to the square root of **x**.

Write a **ConsoleProgram** that accepts values for **a** and **b** as **doubles** (you can assume that **a** and **b** will be positive) and then calculates the solution of **c** as a **double**. Your program should be able to duplicate the following sample run:



4. Write a **ConsoleProgram** that reads in a list of integers, one per line, until a sentinel value of 0 (which you should be able to change easily to some other value). When the sentinel is read, your program should display the smallest and largest values in the list, as illustrated in this sample run:



Your program should handle the following special cases:

- If the user enters only one value before the sentinel, the program should report that value as both the largest and smallest.
 - If the user enters the sentinel on the very first input line, then no values have been entered, and your program should display a message to that effect.
5. Douglas Hofstadter's Pulitzer-prize-winning book *Gödel, Escher, Bach* contains many interesting mathematical puzzles, many of which can be expressed in the form of computer programs. In Chapter XII, Hofstadter mentions a wonderful problem that is well within the scope of the control statements from Chapter 4. The problem can be expressed as follows:

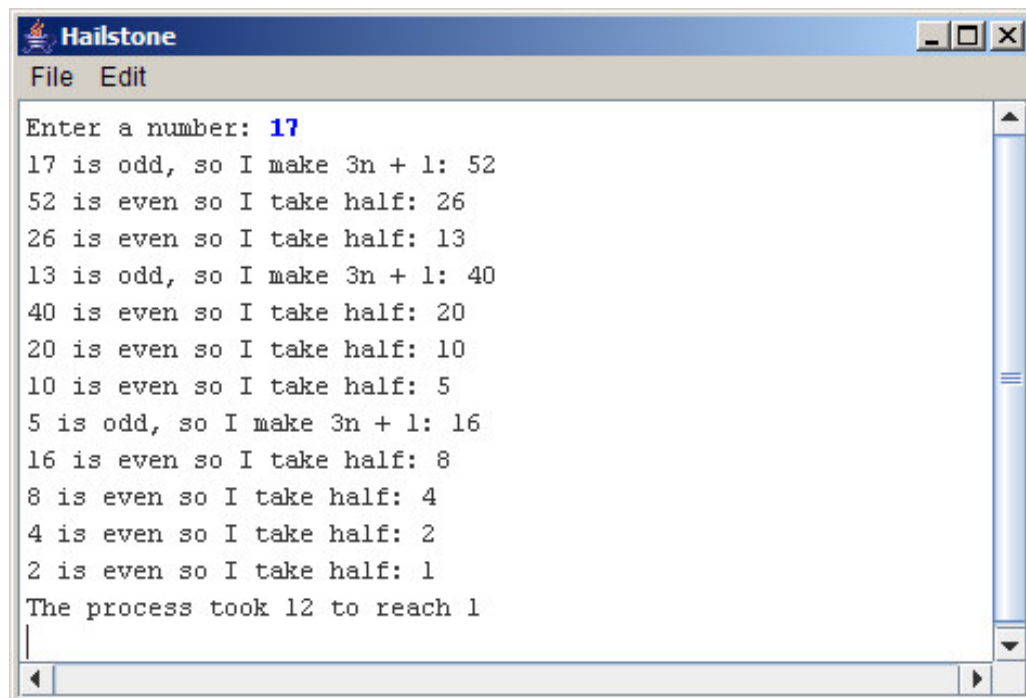
Pick some positive integer and call it n .
 If n is even, divide it by two.
 If n is odd, multiply it by three and add one.
 Continue this process until n is equal to one.

On page 401 of the Vintage edition, Hofstadter illustrates this process with the following example, starting with the number 15:

15	is odd, so I make $3n+1$:	46
46	is even, so I take half:	23
23	is odd, so I make $3n+1$:	70
70	is even, so I take half:	35
35	is odd, so I make $3n+1$:	106
106	is even, so I take half:	53
53	is odd, so I make $3n+1$:	160
160	is even, so I take half:	80
80	is even, so I take half:	40
40	is even, so I take half:	20
20	is even, so I take half:	10
10	is even, so I take half:	5
5	is odd, so I make $3n+1$:	16
16	is even, so I take half:	8
8	is even, so I take half:	4
4	is even, so I take half:	2
2	is even, so I take half:	1

As you can see from this example, the numbers go up and down, but eventually—at least for all numbers that have ever been tried—comes down to end in 1. In some respects, this process is reminiscent of the formation of hailstones, which get carried upward by the winds over and over again before they finally descend to the ground. Because of this analogy, this sequence of numbers is usually called the **Hailstone sequence**, although it goes by many other names as well.

Write a **ConsoleProgram** that reads in a number from the user and then displays the Hailstone sequence for that number, just as in Hofstadter’s book, followed by a line showing the number of steps taken to reach 1. For example, your program should be able to produce a sample run that looks like this:



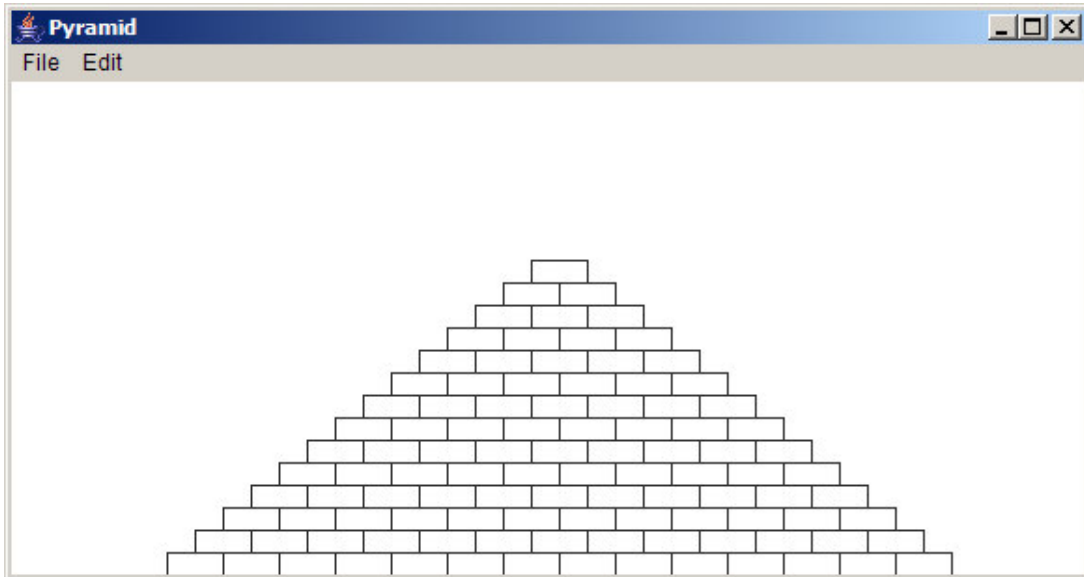
```
Enter a number: 17
17 is odd, so I make 3n + 1: 52
52 is even so I take half: 26
26 is even so I take half: 13
13 is odd, so I make 3n + 1: 40
40 is even so I take half: 20
20 is even so I take half: 10
10 is even so I take half: 5
5 is odd, so I make 3n + 1: 16
16 is even so I take half: 8
8 is even so I take half: 4
4 is even so I take half: 2
2 is even so I take half: 1
The process took 12 to reach 1
```

The fascinating thing about this problem is that no one has yet been able to prove that it always stops. The number of steps in the process can certainly get very large. How many steps, for example, does your program take when n is 27?

Continued on next page...

Graphics Challenges

6. Write a **GraphicsProgram** that draws a pyramid consisting of bricks arranged in horizontal rows, so that the number of bricks in each row decreases by one as you move up the pyramid, as shown in the following sample run:



The pyramid should be **centered** at the bottom of the window and should use constants for the following parameters:

BRICK_WIDTH	The width of each brick (30 pixels)
BRICK_HEIGHT	The height of each brick (12 pixels)
BRICKS_IN_BASE	The number of bricks in the base (14)

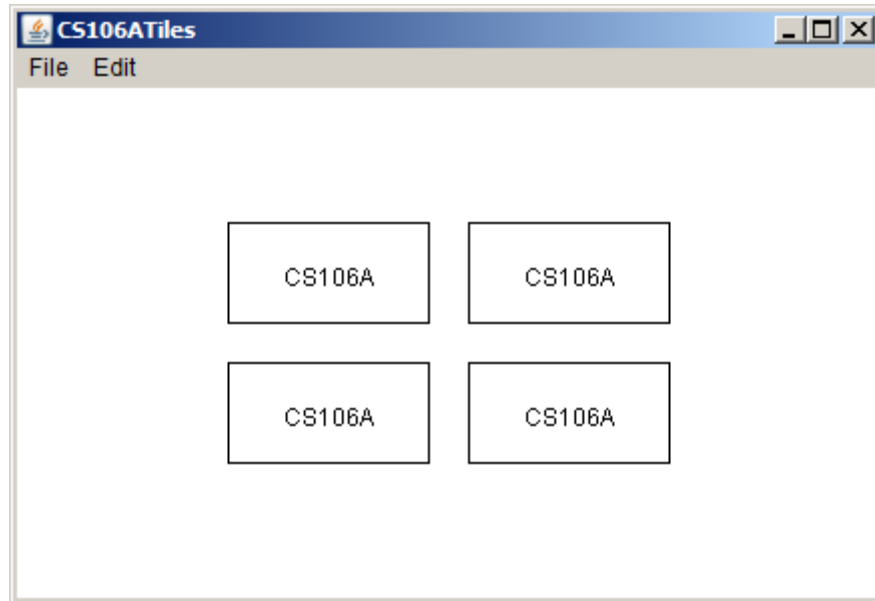
The numbers in parentheses show the values for this diagram, but you must be able to change those values in your program.

7. [Use Methods]. Suppose that you've been hired to produce a program that draws an image of an archery target—or, if you prefer commercial applications, a logo for a national department store chain—that looks like this:



This figure is simply three **G oval** objects, two red and one white, drawn in the correct order. The outer circle should have a radius of one inch (72 pixels), the white circle has a radius of 0.65 inches, and the inner red circle has a radius of 0.3 inches. The figure should be **centered** in the window of a **GraphicsProgram**.

8. [Use Methods]. As an expression of your fondness for CS106A, you should write a **GraphicsProgram** called **CS106ATiles** that display four tiles (rectangles), each containing the text “CS106A” in the center of the graphics window, as shown below:



- The **width** and **height** of each of the four tiles (rectangles) should be specified as named constants **TILE_WIDTH** and **TILE_HEIGHT**, respectively, so that they are easy to change. You should determine reasonable values for these constants to make your picture look similar (but, it need not be exact) to the figure above.
- The text/label “CS106A” should be **centered** in each of the respective four tiles. You can find the width of a label by calling `label.getWidth()` and the height it extends above the baseline by calling `label.getAscent()`. If you want to center a label, you need to shift its origin by half of these distances in each direction.
- The **amount of space** (in pixels) between each of the four tiles is specified by the constant **TILE_SPACE** (which is provided in the starter file). This constant should be used both for the horizontal space between tiles and the vertical space between tiles.
- The entire figure (of four tiles) should be **centered** in the graphics window.

Continued on next page...

Sandcastles

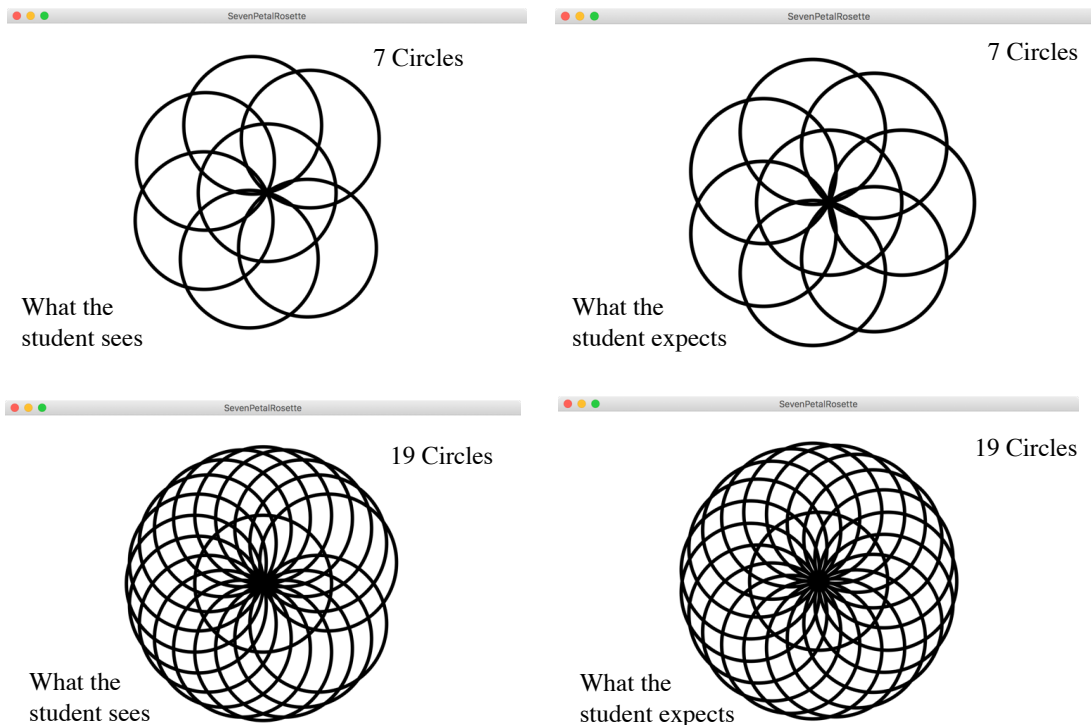
9. Fill in the provided **isDivisibleBy** method with code that returns whether the first parameter **a** is divisible by the second parameter **b**.

We say that **a** is divisible by **b** if and only if (**a** divided by **b**) has remainder 0. You should handle the following edge cases:

- If **a** is less than **b**, you should return false.
- If **a** is ≤ 0 , you should return false.
- Similarly if **b** is ≤ 0 , you should return false.

You don't need to write code in the run method. We have already written code there that you can use to evaluate whether or not your method is functioning correctly.

10. In this problem you are a section leader. Your sectionee has been working on a cool extension program which draws a geometric shape called the Seven Petal Rosette (a variation of a shape which shows up around the world):

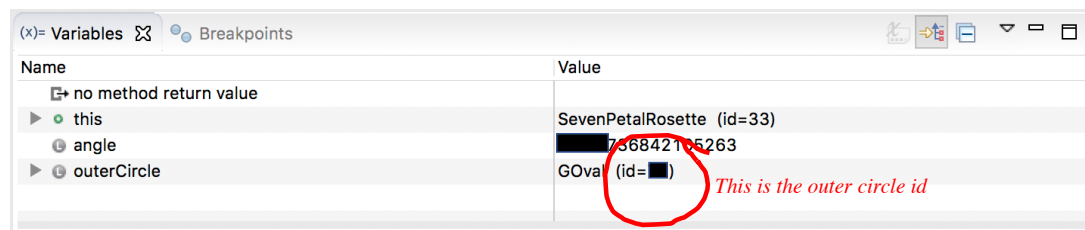


Unfortunately there are bugs in their program and they have asked you for help. In this question we would like you to step through the program one step at a time and report back values that you observe. Identify the changes that your sectionee should make for the program to work correctly.

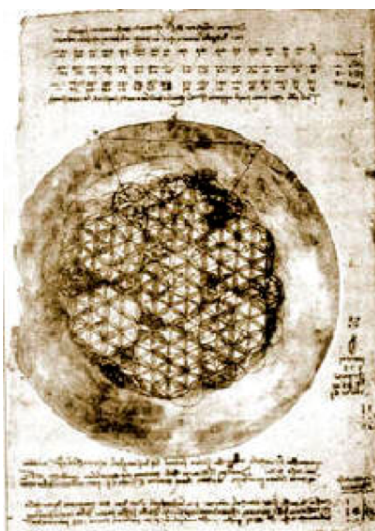
We have included your sectionee’s code in the Assignment 2 project. Above the code in **SevenPetalRosette.java** we have included some specific questions about the sectionee’s program. Write your solution directly into the comments of the code. **Notes:** Where the comments ask for the value of the angle at each iteration, enter the value *after* the angle has been updated during the loop, i.e. after line 61 (`angle += delta`) has executed. The “expected angle” values refer to what one would mathematically expect, but may or may not match the value you will observe when running the program.

You should not make major changes to their code! Your goal is to help them find bugs. You may change the values of constants if you find that helpful and you may test out your suggested bug fixes.

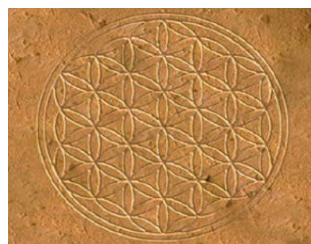
In order to figure out what has gone wrong your first step is to trace through their program using the debugger. When the debugger is running there is a Variables pane which shows you the value and id of all of the variables in the current method. If you open the debugger view (**Stanford Menu → Debugger**, and **Stanford Menu → Editor** to return to the editor view) and don’t see the Variables pane, go to **Window → Show View → Variables** to display it.



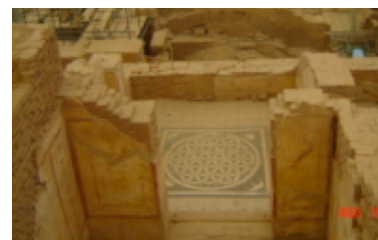
Fun aside. The Six Petal Rosette shows up in ancient art all around the world: (a) Sketch by Leonardo Da Vinci, (b) Abydos, Egypt, (c) Efes, Turkey, (d) Amritsar, India, (e) Nikko, Japan.



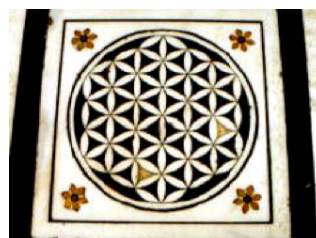
(a)



(b)



(c)



(d)



(e)

That’s all, folks!