Assignment 3: Problem-Solving in Java

Based on an assignment by Eric Roberts and Mehran Sahami

We've just about covered the fundamentals of imperative programming – loops, variables, methods, parameters, and return statements – and now it's time to put them all together! Your task in this assignment is to implement a collection of smaller programs to gain more familiarity and practice with the fundamental concepts you've seen so far.

The starter code for this assignment is available on the CS106A website under the “Assignments” tab.

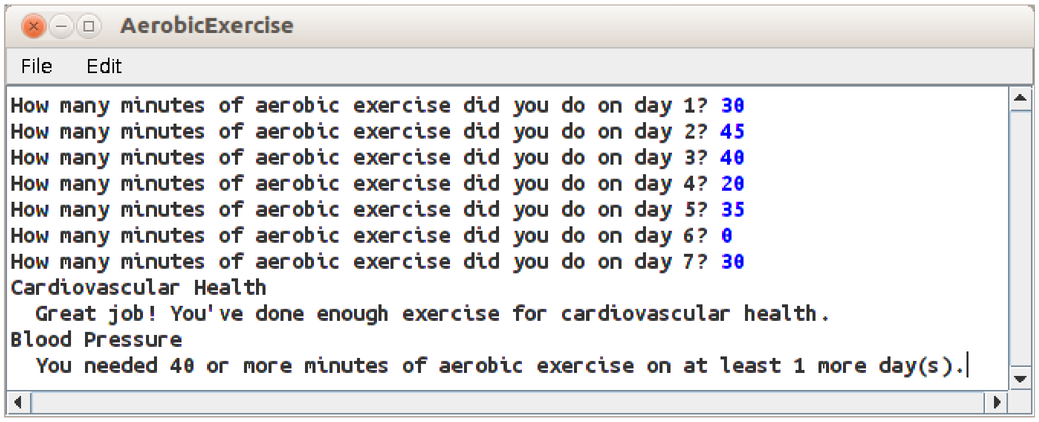
Due Monday, February 2nd at 3:15PM

Part One: Aerobic Exercise

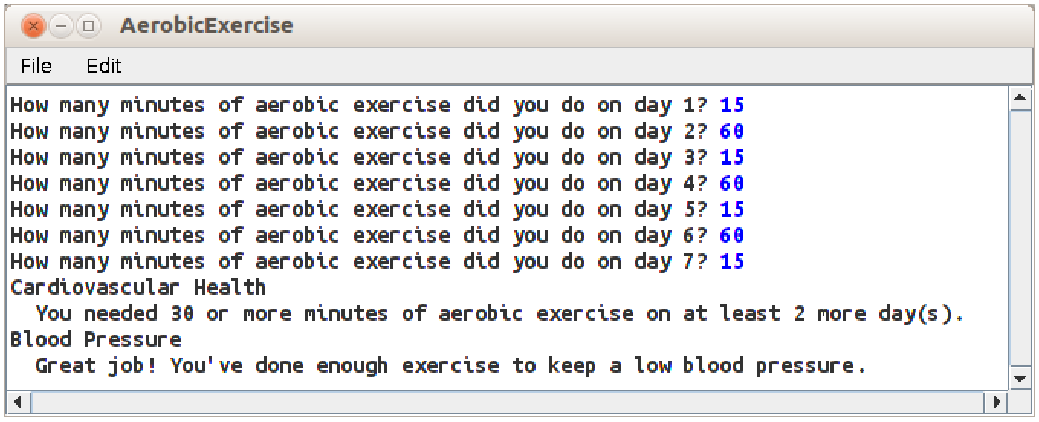
The American Heart Association recommends[*] that you get at least 30 minutes of aerobic exercise on five separate days each week to maintain cardiovascular health. It also recommends that you get at least 40 minutes of aerobic exercise at least three days each week to maintain a low blood pressure. Your task is to write a program that prompts the user for the number of minutes of aerobic exercise they did in each of the last seven days, then to report the following information:

- Whether they did enough exercise to maintain cardiovascular health and, if not, how many more days each week they need to get at least 30 minutes of aerobic exercise.
- Whether they did enough exercise to lower blood pressure and cholesterol and, if not, how many more days each week they need to get at least 40 minutes of aerobic exercise.

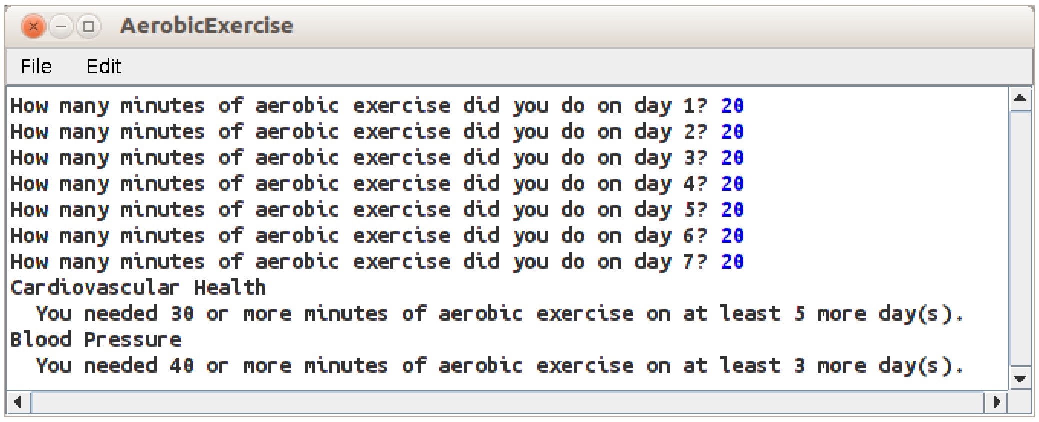
Here are some sample runs of the program:



```
How many minutes of aerobic exercise did you do on day 1? 30
How many minutes of aerobic exercise did you do on day 2? 45
How many minutes of aerobic exercise did you do on day 3? 40
How many minutes of aerobic exercise did you do on day 4? 20
How many minutes of aerobic exercise did you do on day 5? 35
How many minutes of aerobic exercise did you do on day 6? 0
How many minutes of aerobic exercise did you do on day 7? 30
Cardiovascular Health
  Great job! You've done enough exercise for cardiovascular health.
Blood Pressure
  You needed 40 or more minutes of aerobic exercise on at least 1 more day(s).
```



```
How many minutes of aerobic exercise did you do on day 1? 15
How many minutes of aerobic exercise did you do on day 2? 60
How many minutes of aerobic exercise did you do on day 3? 15
How many minutes of aerobic exercise did you do on day 4? 60
How many minutes of aerobic exercise did you do on day 5? 15
How many minutes of aerobic exercise did you do on day 6? 60
How many minutes of aerobic exercise did you do on day 7? 15
Cardiovascular Health
  You needed 30 or more minutes of aerobic exercise on at least 2 more day(s).
Blood Pressure
  Great job! You've done enough exercise to keep a low blood pressure.
```



```
How many minutes of aerobic exercise did you do on day 1? 20
How many minutes of aerobic exercise did you do on day 2? 20
How many minutes of aerobic exercise did you do on day 3? 20
How many minutes of aerobic exercise did you do on day 4? 20
How many minutes of aerobic exercise did you do on day 5? 20
How many minutes of aerobic exercise did you do on day 6? 20
How many minutes of aerobic exercise did you do on day 7? 20
Cardiovascular Health
  You needed 30 or more minutes of aerobic exercise on at least 5 more day(s).
Blood Pressure
  You needed 40 or more minutes of aerobic exercise on at least 3 more day(s).
```

[*] http://www.heart.org/HEARTORG/GettingHealthy/PhysicalActivity/FitnessBasics/American-Heart-Association-Recommendations-for-Physical-Activity-in-Adults_UCM_307976_Article.jsp

Part Two: The Hailstone Sequence

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. Here's one of the puzzles he poses:

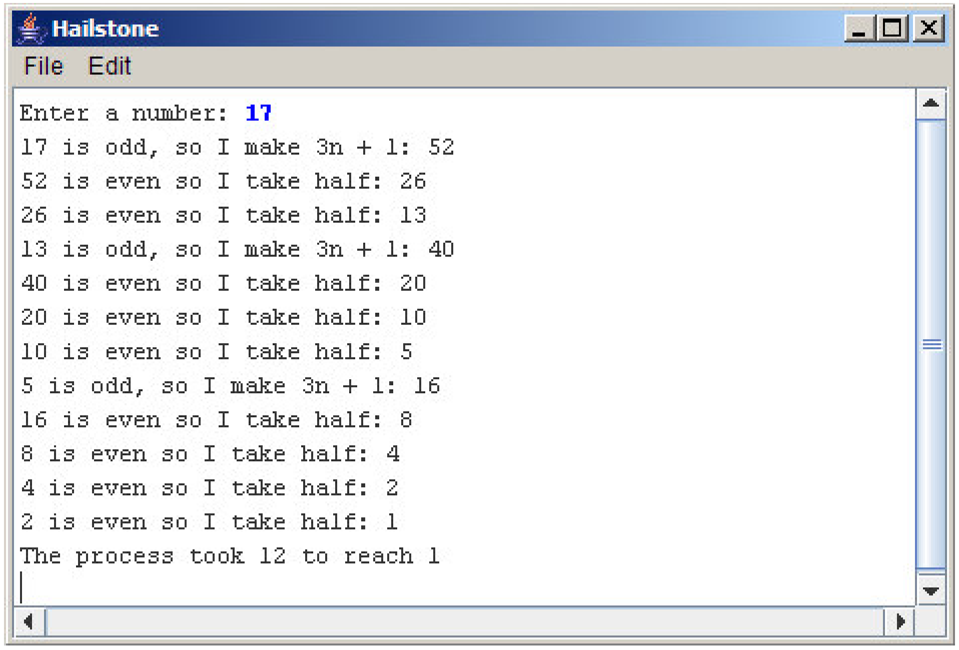
> Pick some positive integer and call it *n*.
>
> If *n* is even, divide it by two.
>
> If *n* is odd, multiply it by three and add one.
>
> Continue this process until *n* is equal to one.

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

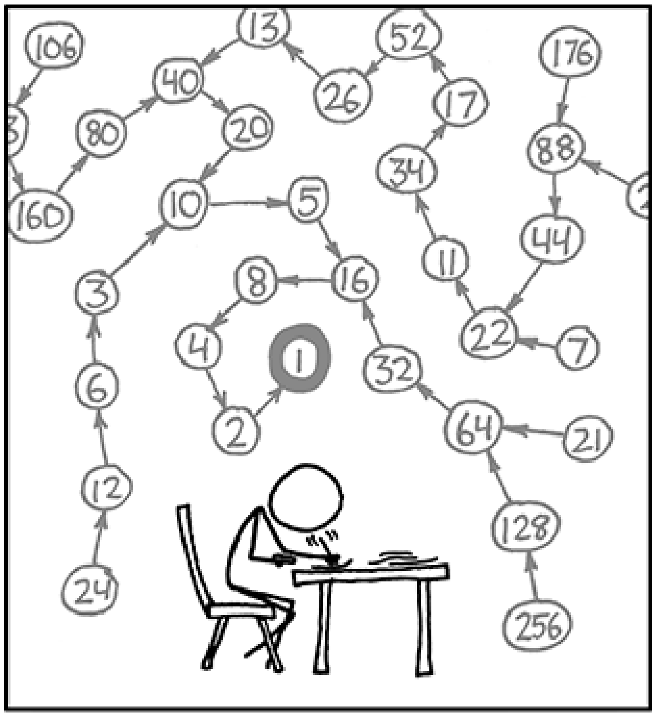
As you can see from this example, the number goes up and down, but eventually—at least for all numbers that have ever been tried—comes down to end in 1. In some respects, this process is reminiscent of the formation of hailstones, which get carried upward by the winds over and over again before they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

Write a program 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 the one on the top of the next page.



```
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 *Collatz conjecture* is that this process always eventually reaches 1. Although the hailstone sequence terminates for all numbers anyone has even tried, no one has yet proven or disproven the Collatz conjecture. If you're interested in learning a bit more about the Collatz conjecture, take CS103!



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT. (xkcd)

Part Three: Exponentiation

Your job in this part of the assignment is to implement a method

```
private double raiseToPower(double base, int exponent)
```

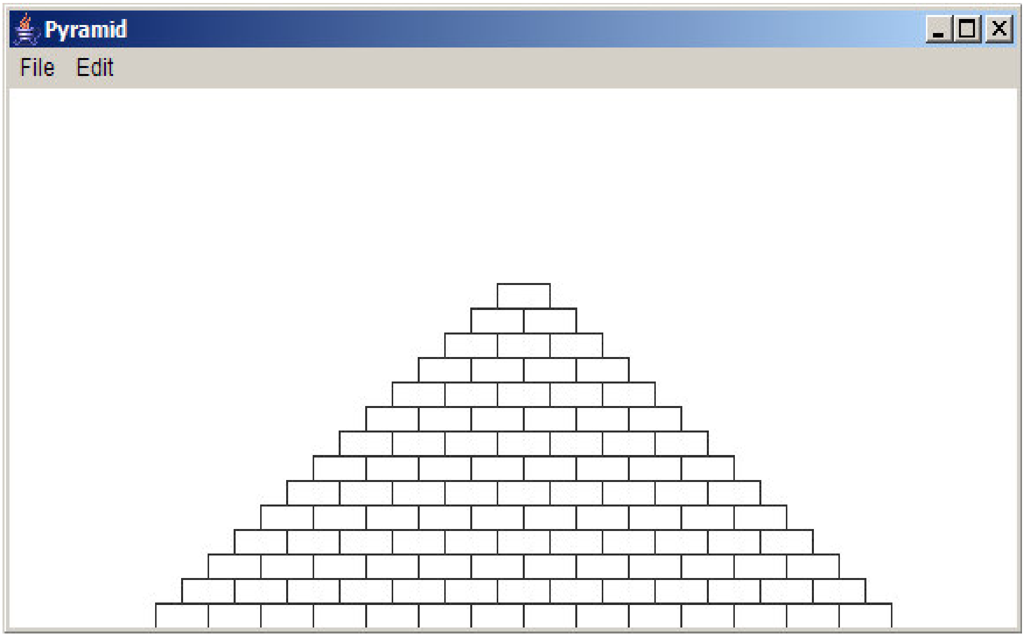
This method will accept two parameters and compute the value of the first parameter raised to the power of the second parameter. For example, calling `raiseToPower(2.0, 3)` would return $2^3 = 8$. Calling `raiseToPower(Math.PI, 2)` would yield $\pi^2 \approx 9.869$. The exponent can be either positive or negative, so calling `raiseToPower(0.5, -2)` should yield $0.5^{-2} = 4$. For the purposes of this assignment, we'll assume that *any* number raised to the zeroth power will be 1. This mean, in particular, that `raiseToPower(0, 0)` should return 1.

Although we didn't discuss this in class, the `double` type actually stores *approximations* of real numbers, so if you work with very large or very small numbers, you may see some inaccuracies in the numbers you work with. That's perfectly fine and is nothing to worry about.

For the purposes of this part of the assignment – and just this part of the assignment – you are not permitted to use the functions `Math.pow`, `Math.exp`, `Math.log`, etc., since they kinda defeat the point of the exercise. ☺

Part Four: Pyramid

In this part of the assignment, we'd like you to write a program that draws a pyramid consisting of bricks arranged in horizontal rows. 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 (by default, 30 pixels)
`BRICK_HEIGHT`	The height of each brick (by default, 12 pixels)
`BRICKS_IN_BASE`	The number of bricks in the base (by default, 14)

Although we've specified default values for these constants, your program should work correctly when the constants are set to arbitrary nonnegative values.

Part Five: The Saint Petersburg Game

The *Saint Petersburg Game* or *Saint Petersburg Lottery* is a hypothetical casino game played by two players (say, you and me) sitting at a table. I begin by putting $1 on the table, and you then repeatedly flip a coin until it comes up tails. Each time the coin comes up heads, I double the amount of money on the table. As soon as the coin comes up tails, the game is over and you win all the money on the table (no strings attached – I'm just feeling really generous!)

One sample run of the game is as follows. I put $1 on the table. You then flip tails, so the game ends and you collect $1. Another run might go like this: I put $1 on the table. You flip heads, so I double the money to $2. You flip heads again, so I double the money to $4. You flip heads again, so I double the money to $8. You then flip tails, so the game ends and you collect $8.

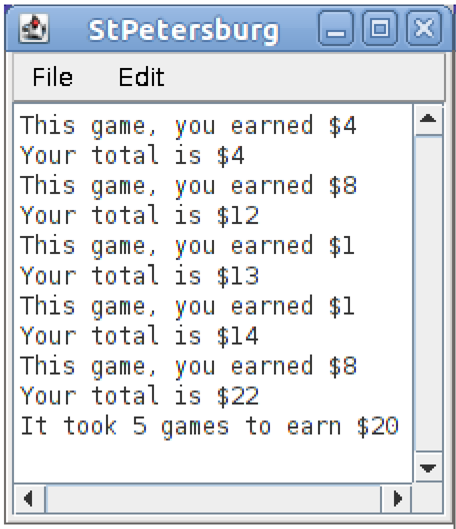
Write a program that plays the Saint Petersburg Game as many times as is necessary to earn a total of at least $20. To clarify – you aren't playing until you win $20 in a single game; instead, you're playing until your total winnings across all games you've played is $20. After each game, your program should print out how much money you won during that game, along with your total winnings. For example, if after the first game you earned $4, you would print out

```
This game, you earned $4
Your total is $4
```

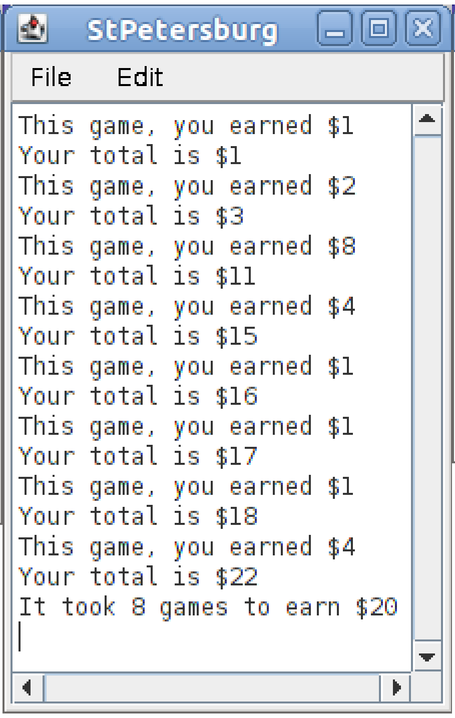
Once you've earned at least $20, your program should output how many times you had to play the Saint Petersburg Game to earn $20. For example, if it takes five games to win, at the end you'd print

```
It took 5 games to earn $20
```

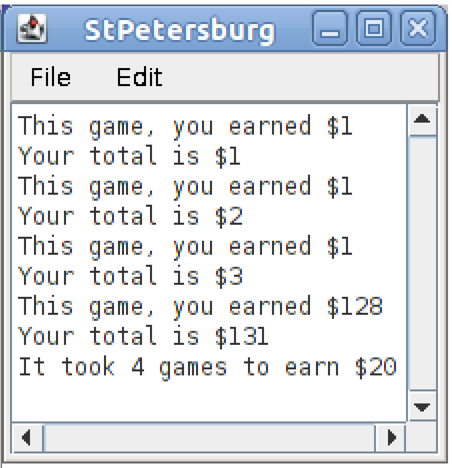
Three sample runs of the program are shown below:



```
This game, you earned $4
Your total is $4
This game, you earned $8
Your total is $12
This game, you earned $1
Your total is $13
This game, you earned $1
Your total is $14
This game, you earned $8
Your total is $22
It took 5 games to earn $20
```



```
This game, you earned $1
Your total is $1
This game, you earned $2
Your total is $3
This game, you earned $8
Your total is $11
This game, you earned $4
Your total is $15
This game, you earned $1
Your total is $16
This game, you earned $1
Your total is $17
This game, you earned $1
Your total is $18
This game, you earned $4
Your total is $22
It took 8 games to earn $20
```



```
This game, you earned $1
Your total is $1
This game, you earned $1
Your total is $2
This game, you earned $1
Your total is $3
This game, you earned $128
Your total is $131
It took 4 games to earn $20
```

The Saint Petersburg game is unusual because the payoffs rise exponentially quickly but become exponentially harder and harder to get. This leads to a very counterintuitive mathematical result – the expected amount of money you should get from playing the Saint Petersburg game just once is *infinite*. Curious to learn why? Take CS109!

Part Six: Five Seconds of Fame

We will cover the topics necessary to complete this problem on Wednesday.

In this last part of the assignment, we'd like you to create a five-second animation of your own choosing! Unlike the previous parts of this assignment, we're not going to tell you what specifically you should animate – that's entirely up to you. We do, however, have the following requirements for your animation:

- Your animation should be roughly five seconds in length. At the end of the quarter, we're planning on showing off the submissions for this assignment, and given the number of students enrolled in CS106A that means that everyone will only get five seconds of screen time.
- Your animation needs to be at least 50 frames long. If you're using a `while` loop to drive your animation at 24 frames a second, you'll immediately satisfy this requirement.

You'll get full credit as long as you submit an animation that meets the above requirements (and isn't copied from one of the lecture examples or section problems). However, we encourage you to be creative. See what you can come up with!

(Optional) Part Seven: Extensions!

Can you think of a way to make one of the programs you wrote more exciting? If so, for extra credit, you're welcome to go above and beyond what we've asked you to do in this assignment.

If you'd like to implement extensions on top of any of these programs, please feel free to do so. To make it easier to grade your assignments, if you do choose to add extensions, please create separate programs for the "base" version of the assignment (what we asked you to do in this handout) and the "extended" version of the assignment (what you chose to do on top.) For example, if you wanted to add extensions to the Aerobic Exercise program, you could submit your assignment containing both an `AerobicExercise.java` program and an `ExtendedAerobicExercise.java` program.

Advice, Tips, and Tricks

Most of the programs in Assignment 2 involved some number of constants, and we advised you to test your programs with lots of different values for those constants. In this assignment, you should continue this tradition, especially in the Pyramid problem.

One major difference between this assignment and the previous assignment is that the majority of these programs respond to user input. Make sure that you test your programs extensively by typing in all sorts of different inputs – it would be a shame if you missed an edge case and didn't notice it during testing.

Style is, as always, very important in these programs. Make sure that, when appropriate, you decompose your programs into smaller, more manageable pieces. For the more complex programs, you'll almost certainly need to introduce helper methods or constants.

Good luck!