Nick Troccoli
CS 106A

Assignment 2
July 5, 2017

Assignment #2: Intro to Java

Due: 11AM PST on Wednesday, July 12

This assignment should be done individually (not in pairs)

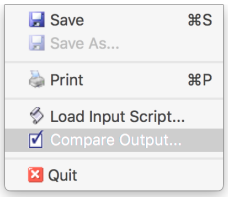
Based on handouts by Mehran Sahami, Eric Roberts and Marty Stepp.

For this assignment, you will write programs to solve five different problems. The starter project, available on the course website, will contain java files for you to write your programs in. Specifically, you will turn in the following files: **`QuadraticEquation.java`**, **`Weather.java`**, **`Hailstone.java`**, **`Rocket.java`** and **`AsciiArt.java`**. You should not modify any other files in the starter project.

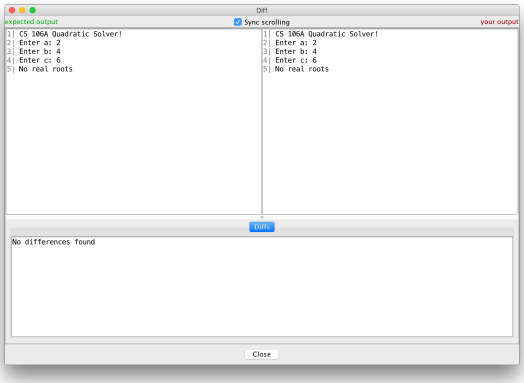
Note that **you should limit yourself to the material covered up until the release of this assignment** (through part of lecture on Wednesday, July 5). You should not use any other material (such as Strings, parameters, instance variables, return, etc.).

Comparing Output: each of your programs must *exactly* match the specified output. To check your output, you can use the *Output Comparison* tool built in to every Console Program. To do this:

1) Run any one of your programs as normal. Click File -> Compare Output…



2) We include several sample output files for each problem; select the output file against which you would like to compare your program. **Note:** the output files are saved in the **output/** folder in your Eclipse project. To view what any output file contains, simply expand the **output/** folder and double-click on any file to open it.
3) A window will appear comparing your program's output with that output file.



Problem 1: Quadratic Equation

Write an interactive `ConsoleProgram` named **QuadraticEquation** that finds real roots of a quadratic equation. A *quadratic equation* is a mathematical equation of the form $ax^2 + bx + c = 0$, where a is nonzero. Given the values of a, b, and c, the quadratic formula says that the roots (values of x) of the quadratic equation are given by:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The quantity (b^2 - $4ac$) is called the *discriminant*. If it's greater than zero, there are two different real roots of the quadratic equation, which are given by the above formula. If it's exactly zero, there's just one root, given in two different ways by the quadratic formula. If it's negative, there are no real roots to the equation.

Your job is to write a program that prompts the user for the values of a, b, and c, then prints out the roots of the corresponding quadratic equation. Your program must *exactly* duplicate the output of the following sample runs below, plus runs with other values. *(User input as read from a call to readInt is shown in blue bold font below.)*

```
CS 106A Quadratic Solver!
Enter a: 1
Enter b: -3
Enter c: -4
Two roots: 4.0 and -1.0
```

```
CS 106A Quadratic Solver!
Enter a: 1
Enter b: 6
Enter c: 9
One root: -3.0
```

```
CS 106A Quadratic Solver!
Enter a: 2
Enter b: 4
Enter c: 6
No real roots
```

expected output from three separate runs of the program (user input in blue)

To compute the **square root** of some number `x`, you can use the `Math.sqrt` method. For example, the following code sets `y` to the square root of `x`:

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

You may assume that the user doesn't enter 0 as their value for a. Aside from the above restriction, the values of a, b, and c can be any integers. You should not do any rounding of real numbers in your output.

While other parts of this homework must be decomposed using **methods**, you aren't required to do so on this problem.

Problem 2: Weather

Write an interactive `ConsoleProgram` named **Weather** that displays information about recent temperatures. The program should repeatedly prompt the user to enter temperatures until a particular "sentinel" value is entered. By default this sentinel value is **-1**. Once the sentinel value is entered, the program should display the maximum and minimum temperature typed, as well as the average temperature and the number of "cold" temperatures entered. Assume that "cold" (extremely subjective) is a temperature of 50 or lower. Your program should *exactly* duplicate the output of the following sample run (user input is shown in blue), plus be able to run properly with other values:

```
CS 106A "Weather Master 4000"!
Next temperature (or -1 to quit)? 68
Next temperature (or -1 to quit)? 94
Next temperature (or -1 to quit)? 76
Next temperature (or -1 to quit)? 45
Next temperature (or -1 to quit)? 89
Next temperature (or -1 to quit)? 36
Next temperature (or -1 to quit)? 73
Next temperature (or -1 to quit)? -1
Highest temperature = 94
Lowest temperature = 36
Average = 68.71428571428571
2 cold day(s).
```

This program should be written to use a **class constant** to represent the sentinel value -1, so that by changing only that constant's value and recompiling / running the program, it will now use the new sentinel value throughout the code and output. For example, if the constant's value is changed to **-42**, the program's output would look like:

```
CS 106A "Weather Master 4000"!
Next temperature (or -42 to quit)? 76
Next temperature (or -42 to quit)? 89
Next temperature (or -42 to quit)? 83
Next temperature (or -42 to quit)? -42
Highest temperature = 89
Lowest temperature = 76
Average = 82.66666666666667
0 cold day(s).
```

If only one temperature is entered, that temperature is the maximum, minimum, and average. For example:

```
CS 106A "Weather Master 4000"!
Next temperature (or -1 to quit)? -10
Next temperature (or -1 to quit)? -1
Highest temperature = -10
Lowest temperature = -10
Average = -10.0
1 cold day(s).
```

If no temperatures are entered, the program should instead print the following message saying that no temperatures were entered:

```
CS 106A "Weather Master 4000"!
Next temperature (or -1 to quit)? -1
No temperatures were entered.
```

While other parts of this homework must be decomposed using **methods**, you aren't required to do so on this problem.

Problem 3: Hailstone

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 can be expressed as follows:

> Pick some positive integer and call it *n*.
> Do the following process until *n* is equal to one:
> - If *n* is odd, multiply it by three and add one.
> - If *n* is even, divide it by two.

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. Then, after each hailstone sequence, the program should ask the user if they would like to enter another number. You may assume the user will type exactly "y" or "n" to this question. The program should continue outputting the hailstone sequence for the number the user enters until the user chooses not to enter another number. For example, your program should *exactly* duplicate the output of the following sample run (user input is shown in blue), plus be able to run properly with other values:

```
This program computes Hailstone sequences.

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
It took 12 steps to reach 1.
Run again? y

Enter a number: 4
4 is even, so I take half: 2
2 is even, so I take half: 1
It took 2 steps to reach 1.
Run again? y

Enter a number: 1
It took 0 steps to reach 1.
Run again? n
Thanks for using Hailstone.
```

As shown above, you should handle the special case where the user enters 1, and output that it took 0 steps. You can **ask the user to play again** using the `readBoolean` method directly in an if statement or while loop, without storing it in a variable first. For example:

```
while (readBoolean("Run again? ", "y", "n")) {
```

We recommend starting by writing code to output a single Hailstone sequence and test this thoroughly before trying to handle the "Run again?" aspect. This program must have a method to output a single Hailstone sequence *("Enter a number ..." through "It took ..." inclusive). **Note**: using a method that calls itself instead of using loops is not appropriate for this problem, and will result in a deduction.*

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?

Problem 4: Rocket

Write a (non-interactive) `ConsoleProgram` named **Rocket** that displays a specific text figure that looks like a rocket ship. You must **exactly** reproduce the format of the output at right, including identical characters and spacing. You must use **nested for loops** for lines that have repeated patterns of characters that vary in number from line to line, rather than using a single println statement that prints each line of the figure.

Figure at size 5:

```
CS 106A Rocket
(size 5)
     /\
    //\\
   ///\\\
  ////\\\\
 /////\\\\\
+==========+
|..../\....|
|.../\/\...|
|../\/\/\..|
|./\/\/\/\.|
|/\/\/\/\/\|
|\/\/\/\/\/|
|.\/\/\/\/.|
|..\/\/\/..|
|...\/\/...|
|....\/....|
+==========+
     /\
    //\\
   ///\\\
  ////\\\\
 /////\\\\\
```

Constant: Another significant component of this assignment is the task of generalizing the program using a single **constant** that can be changed to adjust the size of the figure. You should create one *(and only one)* constant named SIZE to represent the size of the pieces of the figure. Use **5** as the value of your SIZE constant. Your figure must be based on that exact value to receive full credit.

However, on any given execution your program will produce just one version of the figure. But by simply changing your constant's value and recompiling, your program would produce a figure of a different size. Your program should scale for any constant value of 2 or greater. For example, below at right is the output your program should produce at a SIZE of 3. You can use the previously-mentioned output comparison tool to check your output with various size constant values.

Figure at size 3:

```
CS 106A Rocket
(size 3)
   /\
  //\\
 ///\\\
+=======+
|../\..|
|./\/\.|
|/\/\/\|
|\/\/\/|
|.\/\/.|
|..\/..|
+=======+
   /\
  //\\
 ///\\\
```

Methods: You must use **methods** to represent each part of the rocket, structuring your solution in such a way that the methods match the structure of the output itself. Avoid significant redundancy; use methods so that no substantial groups of identical statements appear in your code. In this problem, **no println statements should appear in your run method**. You do not need to use methods to capture redundancy in partial lines, such as if a single line has the substring ... printed twice.

Development Strategy: We suggest that you not worry about the constant at first. Write an initial program without a constant that produces the default size-5 output. Make sure you use nested for loops for sequences of repeated characters. After your figure looks correct at the default size, then modify the code to use the constant.

Problem 5: Ascii Art

For this problem, write a `ConsoleProgram` named **AsciiArt** that produces any text art (sometimes called "ASCII art") picture you like, with the following restrictions and details. Be creative! *This problem is not graded on Style - as long as your AsciiArt program compiles, runs and meets these constraints, it will receive full credit.*

Constraints:

- The art should be your own creation, not an ASCII image you found elsewhere.
- The art should be 3-200 lines total, with no more than 200 characters per line.
- The art should not include hateful, offensive, or otherwise inappropriate content.
- The code should use at least one loop or method, but it should not have any infinite loops and should not read any user input (e.g. readInt).
- The art must not be completely taken from your solution to any other part of this assignment.
- You must **display your name** somewhere in the output. The simplest way to do this would be to println a line at the start saying something like, "CS 106A ASCII Art by John Smith and Jane Doe".

For example, here is a possible ASCII art of a smiley face (though you should make up your own output and not match this one exactly):

```
CS 106A ASCII Art by Randy Gardner
            /\
           /\/\/\
          /\/\/\/\/\
         /\/\/\/\/\/\/\
           .-'""""""'-.
         .'            `.
        /   .      .    \
       :                  :
       |                  |
       :   \        /     :
        \   `.____.'      /
         `.              .'
           `-._____.-'
```

Neat console features: If you want to have some fun, there are a few features of the console that you may want to use. One is the ability to print in colors. To print in color, pass a second parameter to the println or print method that indicates a color such as RED, GREEN, or BLUE, such as the following:

```
println("Hello!", Color.BLUE);      // print blue text
```

Another fun feature you may want to use is to "animate" your ASCII art by printing some text, then clearing the console by calling

```
pause(x);                  // pauses for x milliseconds
clearConsole();            // clears console window
```

and then printing some text again. This will produce an appearance that looks like animation. Be creative!

Development Strategy and Grading

As with the last assignment, it helps to have a step-by-step process, or **development strategy**, for solving it. Rather than trying to write an entire program without running or testing it, we suggest an **incremental approach**: Try solving a small part of the problem, running it to verify that what you wrote works properly, then continue.

Functionality: Your code should compile without any errors or warnings. On this assignment, the console programs must work for a variety of **user inputs**. For example, the weather program should work for any integer temperature the user types. As with the Karel assignment, test your programs extensively before submitting them.

Some of the other programs that you'll be writing need to refer to **constants** defined in your program. A constant makes it easier to change your program's behavior simply by adjusting the value assigned to that constant. When grading, we will run your programs with a variety of different constant values to test whether you have correctly and consistently used constants throughout your program. Before submitting, you should check whether or not your programs work when you vary the values of the constants as appropriate.

Style: Style is just as important as ever in this assignment. Have you read the **Style Guide** on the course website yet? If not, please do so (it's in the "Assignments" tab). It will be updated for each assignment with any new constraints to be aware of. Be sure to still follow the guidelines laid out in the Karel assignment, as well as any new ones added to the online guide or shown in class.

Procedural decomposition: The QuadraticEquation, Weather, and AsciiArt programs do not require methods, but for the other problems, your run method should represent a concise summary of the overall program, calling other methods to do the work of solving the problem, but run itself should not directly do much of the work.

Fields / instance variables: You should never declare any "global" variables outside of methods (also called "instance variables"). Always declare local variables that exist only inside a single method. If you need to use a value between multiple methods, declare a constant instead or rethink your decomposition.

Honor Code: Remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also do not give out your solution and do not place your solution on a public web site or forum. Remember that all solutions from this quarter and past quarters, as well as any solutions found online, will be electronically compared. If you need help, please seek out our available resources to help you; we are more than happy to try to help you solve these problems.