Stanford University, CS 106A, Homework Assignment 2
Consolation Prize (Java Console Programs) (PAIR ASSIGNMENT)

Based on past assignments created by Keith Schwarz, Eric Roberts, Mehran Sahami, Stuart Reges, etc. Modified by Marty Stepp.

This assignment practices standard Java programming such as console interaction, variables, constants, control statements, parameters, and graphics. As with all assignments, there is a **starter project ZIP archive** including all of these problems on our web site in the area for Homework 2. You will **turn in** the following files:

- **QuadraticEquation.java**, **ExamScores.java**, **GuessingGame.java**, **Rocket.java**, and **AsciiArt.java**

The ZIP archive contains other files and libraries; you should not modify these. When grading/testing your code, we will run your **.java** code with our own original versions of the support files, so your code must work with them.

This is a **pair assignment**. You may work in a pair, or you may work individually. If you're **looking for a partner**, you can try to meet one in your section, or we have a partner search thread on the course message board.

Comparing Output: We are picky about output matching exactly! To check your output, use the Output Comparison Tool link on the class web site and copy/paste the output from your program. Or just click File → Compare Output... in your program's console window and select one of the provided example output files.

Turnin: If you work as a pair, **comment both members' names** on top of every .java file. **Only one of you should submit** the assignment; do not turn in two copies. Submit using the Submit Project entry in the Stanford Menu in Eclipse.

Problem 1: `QuadraticEquation`

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

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

The quantity (b^2 - 4 a c) 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 Part 1.

Problem 2: ExamScores

Write an interactive `ConsoleProgram` named **`ExamScores`** that displays information about exam scores. The program should repeatedly prompt the user to enter exam scores until a particular "sentinel" value is entered. By default this sentinel value is `-1`. You may assume that the user will type positive integers other than the sentinel itself. Once the sentinel value is entered, the program should display the maximum and minimum exam score typed, as well as the average score and the number of scores that failed the exam. "Failing" the exam in this context is receiving a score of 59 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 "Exam Master 4000"!
Next exam score (or -1 to quit)? 68
Next exam score (or -1 to quit)? 94
Next exam score (or -1 to quit)? 76
Next exam score (or -1 to quit)? 45
Next exam score (or -1 to quit)? 89
Next exam score (or -1 to quit)? 54
Next exam score (or -1 to quit)? 73
Next exam score (or -1 to quit)? -1
Highest score = 94
Lowest score = 45
Average = 71.28571428571429
2 student(s) failed the exam.
```

This program should be written to use a **class constant** to represent the sentinel value of -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 "Exam Master 4000"!
Next exam score (or -42 to quit)? 76
Next exam score (or -42 to quit)? 89
Next exam score (or -42 to quit)? 83
Next exam score (or -42 to quit)? -42
Highest score = 89
Lowest score = 76
Average = 82.66666666666667
0 student(s) failed the exam.
```

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

```
CS 106A "Exam Master 4000"!
Next exam score (or -1 to quit)? 83
Next exam score (or -1 to quit)? -1
Highest score = 83
Lowest score = 83
Average = 83.0
0 student(s) failed the exam.
```

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

```
CS 106A "Exam Master 4000"!
Next exam score (or -1 to quit)? -1
No scores were entered.
```

Problem 3: GuessingGame

Write an interactive `ConsoleProgram` named **`GuessingGame`** that allows the user to play a game in which the program thinks of a random integer and accepts guesses from the user until the user guesses the number correctly. After each incorrect guess, you will tell the user whether the correct answer is higher or lower.

```
CS 106A Guessing Game!

Guess my number from 1-100.
Your guess? 50
It's lower.
Your guess? 25
It's higher.
Your guess? 35
It's lower.
Your guess? 30
It's higher.
Your guess? 31
You got it in 5 guesses!
Play again? y

Guess my number from 1-100.
Your guess? 64
You got it on the first try!
Play again? y

Guess my number from 1-100.
Your guess? 60
It's lower.
Your guess? 20
It's higher.
Your guess? 30
It's higher.
Your guess? 40
It's higher.
Your guess? 50
It's lower.
Your guess? 47
It's higher.
Your guess? 49
You got it in 7 guesses!
Play again? n
Thanks for playing.
```

The log at left shows one sample execution of your program. Your output will differ depending on the random numbers chosen and user input typed, but the overall output structure should *exactly* match that shown below.

Next, a series of guessing games is played. In each game, the computer chooses a random number between 1 and 100 inclusive. The game asks the user for guesses until the correct number is guessed. After each incorrect guess, the program gives a clue about whether the correct number is higher or lower than the guess. Once the user types the correct number, the game ends and the program reports how many guesses were needed.

After each game ends and the number of guesses is shown, the program asks the user if he/she would like to play again. You may assume the user will type exactly "y" or "n" to this question. Once the user chooses not to play again, the program exits. You can **ask the user to play again** using the `readBoolean` method in an `if` statement or `while` loop, such as:

```
if (readBoolean("Play again? ", "y", "n")) {
```

As shown at left, you should handle the special case where the user guesses the correct number on the first try.

Produce repetition using **while** loops. You may also want to review fencepost loops and sentinel loops from lecture. *(Some students avoid properly using while loops by writing a method that calls itself; this is not appropriate on this assignment and will result in a deduction.)*

Your game should produce different randomly chosen numbers every time it is run. Produce random numbers using the `RandomGenerator` as shown in lecture.

Development Strategy: Begin by writing code to play a single guessing game and test this thoroughly before trying to handle the "Play again?" aspect. We suggest temporarily adding a `println` at the start of the game that tells the player the correct number, so you can debug your logic while knowing the right answer.

Method: This program must have a method to play a single game *("Guess my number ..." through "You got it ..." inclusive)*. Your `run` method should call this method repeatedly in a `while` loop to represent the overall program behavior.

Constant: This program should be written to use a **class constant** for the maximum number used in the games. The log above shows games from 1 to 100, but you should be able to change the constant value to use other ranges such as from 1 to 50, or 1 to 1000, or any maximum. The log below at left shows a partial example with max 5.

```
Guess my number from 1-5.
Your guess? 2
It's higher.
Your guess? 4
It's lower.
Your guess? 3
You got it in 3 guesses!
```

Use your constant throughout your code and do not refer to the number 100 directly. Test your program by changing your constant and running it again to make sure that everything uses the new value. A guessing game for numbers from 1 to 5 would produce output such as that shown at left. The web site shows other expected output cases. We suggest that you add the constant last, after all of the other code works properly.

Problem 4: `Rocket`

Write a `ConsoleProgram` named **`Rocket`** that displays a specific text figure that is supposed to look like a rocket ship. *(Use your imagination.)* Unlike the others, this is not an interactive program; it does not read any input from the user. You must **exactly** reproduce the format of the output at right, including identical characters and spacing.

Figure at size 5:

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

Nested Loops: One way to write a Java program to draw this figure would be to write a single `println` statement that prints each line of the figure. However, this solution would not receive full credit. A major part of this assignment is showing that you understand **nested `for` loops**, methods and class constants.

In lines that have repeated patterns of characters that vary in number from line to line, represent lines and character patterns using nested `for` loops. *(See lecture slides.)*

Constant: Another significant component of this assignment is the task of generalizing the program using a single **class constant** that can be changed to adjust the size of the figure. You should create one *(and only one)* class 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.

On any given execution your program will produce just one version of the figure. However, you should refer to the class constant throughout your code, so that 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.

Figure at size 3:

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

The course web site will contain files that show you the expected output if your size constant is changed to various other values. You can use our **Output Comparison Tool** on the course web site (or click File -> Compare Output... in your program's console window) to measure numbers of characters and to verify the correctness of your output for various values of the size constant.

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.

If you like, you may also use Java features from Chapter 5 such as parameters, although you are not required to do so and will receive no extra credit for doing so.

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, begin modifying the code to use the constant.

Problem 5: `AsciiArt`

For this problem, turn in a `ConsoleProgram` named **`AsciiArt`** that produces any text art (sometimes called "ASCII art") picture you like. Your program can produce any text picture you like, with the following restrictions and details.

Constraints:

- The picture should be your own creation, not an ASCII image you found on the Internet or elsewhere.
- The picture should contain between 3-200 lines total, and no more than 200 characters / line.
- The picture should not include hateful, offensive, or otherwise inappropriate images.
- 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 picture 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".
- You can use any advanced features of Java you want, on this part of the assignment only.

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 with this one, 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(milliseconds); , then clearConsole(); , then printing some text again. This will produce an appearance that looks like animation. Be creative!

Pairs: If you are working in a pair and each of you want to draw your own unique ASCII art, you can submit an AsciiArt and AsciiArt2 containing each of your drawings. But you are not required to make two of them and will get full credit for turning in just one drawing. If you turn in one drawing, make sure to print both partners' names at the start of its output, such as, "CS 106A ASCII Art by John Smith and Jane Doe".

Grading: If your `AsciiArt` program compiles and runs and meets the above constraints, it will receive full credit for this part. This part will **not be graded on style**. The goal here is to let you play around a bit while allowing you to be creative and make something neat of your own, rather than having all of the assignment be rigidly specified.

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: We will grade each program's behavior to give you a **Functionality score**. 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 guessing game should work for any positive integer the user types. As with the Karel assignment, test your programs extensively before submitting them. It would be a shame if your section leader dropped you from a ✓+ to a ✓ because you had forgotten to test your program on some particular input.

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 course **Style Guide** yet? If not, please do so. 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.

There are many general Java coding styles that you should always follow. Use descriptive **names** for variables and methods; for example, in general you should avoid one-letter variable names for all variables other than `for` loop counter variables. **Format** your code cleanly using proper indentation and whitespace (use Eclipse "Format" feature). Avoid **redundancy** by using methods, loops, `if/else` factoring, and other techniques taught in class.

In general, limit yourself to using Java syntax taught in lecture and the parts of the textbook we have read so far.

Procedural decomposition: The `QuadraticEquation`, `ExamScores`, and `AsciiArt` 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. Instead, `run` should call other methods to achieve the overall goal.

Each method should perform a single clear, coherent task. No one method should do too large a share of the overall work. As a rough estimate, a method whose body (excluding the header and closing brace) has more than 25 lines is too large. Your methods should also help you avoid **redundant code**. If you are performing identical or similar commands repeatedly, factor out the common code into a method, or otherwise remove the redundancy.

Fields / instance variables: You should never declare any "global" variables outside of methods (also called "fields" or "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.

Commenting: Your programs should have adequate **commenting**. The top of each .java file should have a descriptive comment header with your name, a description of the problem you are solving, and a **citation of all sources** you used to help write your program. Each method you write should have a comment header describing its behavior. For larger methods, you should also place a brief inline comment on any complex sections of code to explain what the code is doing. See the programs from lecture and Style Guide for examples of proper commenting.

Honor Code: Remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions (outside of your pair, if you are part of one). 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. Please be careful. 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.