Programming Assignment #3: Strands

COP 3330, Fall 2019

Due: Wednesday, October 16, *before* 11:59 PM

Abstract

This programming assignment will give you a playground in which to hone your problem solving skills while also gaining experience working with and manipulating strings in Java.

You will write a series of related methods that will require you to employ thoughtful logic in order to produce particular patterns as you print the characters of a string – or, in some cases, multiple strings.

The required methods in this assignment are arranged in order of difficulty (from least difficult to most difficult) and are designed to build upon one another in a way that will help lead you to the solutions to these progressively more sophisticated puzzles. Some of those jumps might still be quite challenging, though.

This is also our first assignment this semester to focus a bit on runtime efficiency. Namely, there are a few restrictions related to slow string processing operations.

Deliverables

Strands.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Preliminaries

1.1. Avoid String Concatenation in Java

For this assignment, you'll be working extensively with strings in Java. You might want to familiarize yourself with the methods in [Java's *String* class](). There's a constructor that produces a *String* from a *char* array. Neat!

(**Special Restriction!**) We've discussed in class that string concatenation in Java is slow. You must avoid using it wherever possible. For example, the following loop would be catastrophically slow if it were operating on a very long string:

```java
public static void printUpperCaseString(String str)
{
    String modifiedString = "";

    for (int i = 0; i < str.length(); i++)
        // This slow concatenation operation is called repeatedly! Yikes!
        modifiedString += Character.toUpperCase(str.charAt(i));

    System.out.println(modifiedString);
}
```

Throughout this assignment – particularly in the later methods that you write – you will need to construct lines of output to print to the screen. To do this, you might be tempted to create a new string for each line of output, concatenate individual characters to that string one by one, and then print the final result before moving on to the next line. However, that act of repeated concatenation is strictly forbidden in this assignment. If you want to add to the end of a string, one of the most straightforward ways to do that without string concatenation in Java would be to simply create a *char* array and fill it in with characters from left to right, then print the contents of the array to the screen. In doing so, you would want to predetermine how long the *char* array needs to be in order to accommodate the longest line of output for a given method call. For many of the methods below, the length of that longest line of output varies depending on what strings are passed as parameters. It will always be possible to determine the maximum output line length before printing anything to the screen, though – either based on the parameters passed to a method, or based on a predetermined limit derived from a method's limited behaviors.

It's actually possible to complete this assignment without creating any new *String* or *char[]* variables (other than the ones passed to the methods you're required to write), but the logic to do so in some of these methods will be exceedingly complicated if you take that route.

1.2. Suggested Helper Method

When writing up this program, I personally found it helpful to write a helper method that took a *char* array as its only parameter and printed everything in the array *except* any and all spaces at the end of the array. So, given the array {'h', 'i', ' ', ' ', ' '}, my helper method printed "hi" (without the quotes), followed by a newline character.

1.3. Output Format

All of the method descriptions below are very precise, but fully comprehending those descriptions (and making some necessary inferences) might take significant cognitive effort. When in doubt about the expected format for any method's output, please refer to the sample test case output files included with this assignment. Note that those might look wonky in older versions of Notepad, which would display the file contents on one long line.

2. Method and Class Requirements

Implement the following methods in a public class named *Strands*. Please note that they are all **public** and **static**. You may implement helper methods as you see fit. Please include your name, the course number, the current semester, and your NID in a header comment at the very top of your source file.

```
public static void straightAndNarrow(String str)
```

Description: This method takes a single string (*str*) and prints the string vertically, with each character from *str* preceded by two spaces and followed by a single newline character. For example, *straightAndNarrow("linear")* should produce the following output. Notice that each character is preceded by two spaces:

```
  l
  i
  n
  e
  a
  r
```

If *str* is *null* or an empty string, this method should simply print "Pshh!" to the screen (without the double quotes), followed by a newline character. For example:

```
Pshh!
```

Note that if a space character appears in *str*, you should simply print a blank line in its place. For example, in the output for *straightAndNarrow("no way")*, the third line would contain a single newline character, and nothing else (no spaces):

```
  n
  o

  w
  a
  y
```

Parameter Restrictions: While *str* might contain space characters (' '), it will not contain any other whitespace characters (tabs, newline characters, etc.). Note that *str* might be *null*.

Output: See above for a description of this method's output. Also, see the test cases and sample output included with this assignment for further examples of this method's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* method and therefore should not return a value.

```
public static void straightAndNarrow(String str, int n)
```

Description: (Note: This is an overloaded method.) This method takes a string parameter (*str*) and an integer (*n*) and prints the string vertically as described in the previous version of this method, but with the following twist: each character is now preceded by *n* spaces (rather than two spaces). All other requirements and restrictions from the previous version of this method apply to this version, as well.

For example, *straightAndNarrow("linear", 0)* should produce the following output. Notice there are zero spaces preceding each character:

```
l
i
n
e
a
r
```

Similarly, *straightAndNarrow("linear", 5)* should produce the following output. Here, there are five spaces preceding each character:

```
     l
     i
     n
     e
     a
     r
```

Parameter Restrictions: While *str* might contain space characters (' '), it will not contain any other whitespace characters (tabs, newline characters, etc.). Note that *str* might be *null*. We guarantee we will never pass a negative integer to this method for *n*.

Output: See above for a description of this method's output. Also, see the test cases and sample output included with this assignment for further examples of this method's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* method and therefore should not return a value.


```
public static void meanderingPath(String str)
```

Description: This method has the same behavior as the *straightAndNarrow(String str)* method, except characters are indented according to the following pattern:

- The following sequence indicates how many spaces should appear before each successive character in the output of this method: 2, 2, 2, 3, 4, 5, 5, 5, 4, 3. From there, this spacing sequence repeats.

- There is a special exception to the sequence given above: the very first character in the string, rather than being preceded by just two spaces, should always be preceded by three spaces. So, the first 2 in the pattern given above is replaced with a 3 for the first character of the string only.

- As before, if a space character appears in *str*, you should simply print a blank line (with no spaces whatsoever).

For example, *meanderingPath("**************")* (with 14 asterisks) should produce the following output. Notice that the 2-2-2-3-4-5-5-5-4-3 indentation pattern starts to repeat after the first 10 characters are printed. Notice also that the first line deviates from that pattern; that line begins with three spaces instead of two:

```
   *
  *
  *
   *
    *
     *
     *
     *
    *
  *
  *
  *
  *
   *
```

Similarly, consider *meanderingPath("*** ***")*. Typically, the fourth line of output would begin with 3 spaces, and the fifth line of output would begin with 4 spaces. However, since our fourth character in that string is a space, the fourth line of output will be blank. This does **not** affect the number of spaces leading the fifth line of output, though; we still proceed to use 4 spaces at the beginning of that fifth line. The output is as follows:

```
   *
  *
  *

    *
     *
     *
```

Parameter Restrictions: While *str* might contain space characters (' '), it will not contain any other whitespace characters (tabs, newline characters, etc.). Note that *str* might be *null*.

Output: See above for a description of this method's output. Also, see the test cases and sample output included with this assignment for further examples of this method's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* method and therefore should not return a value.

```
public static void classicRapunzel(String [] strings)
```

Description: This method receives an array of strings and prints them out side by side using the same indentation pattern described in the *meanderingPath(String str)* method, but with the following modifications:

- Within a given line of output, the characters from each successive string that we're printing should be separated by two spaces.

- If we're printing a line of output and some earlier string in the array (*s1*) has run out of characters, but a string after it (*s2*) has not, we will print a space for the *s1* string's character on that line so as not to disrupt the continuity of the output pattern for *s2*.

- However, we should never print any spaces after the last visible character on any given line of output.

For example, consider the output for *classicRapunzel(new String[] {"*****", "@@", "+++"})* . We already know the pattern for printing the first string (five asterisks):

```
  *
 *
 *
  *
   *
```

Here's what that looks like when we add the second string (two '@' symbols). Notice that each '@' symbol in the second string is preceded by exactly two spaces:

```
  *   @
 *   @
 *
  *
   *
```

Here's what that looks like when we add the third string (three '+' symbols). Notice that each '+' symbol in the third string is preceded by exactly two spaces.

```
  *   @   +
 *   @   +
 *       +
  *
   *
```

In the output above, there are a few additional things to note. Firstly, note that there are **<u>no spaces</u>** after the single asterisk on each of the last two lines. Secondly, notice that on the third line, since the second string had already run out of characters, we had to print a space instead. (In other words, we printed a space directly below the final '@' symbol.) Without that space, the final '+' in the third string would be dragged over to the left a bit too far and wouldn't line up correctly with the other '+' symbols in that string.

Let's consider another call to this method:

classicRapunzel(new String[] {"clandestine", "counterintuitive", "swirlydurly"})

The output for that call is as follows:

```
 c   c   s
 l   o   w
 a   u   i
  n   n   r
   d   t   l
     e   e   y
     s   r   d
     t   i   u
    i   n   r
   n   t   l
 e   u   y
     i
     t
      i
        v
         e
```

Notice that once "clandestine" runs out, we continue leaving spaces so that the "-itive" in "counterintuitive" is still properly attached to that string. However, the "-itive" lines at the end of "counterintuitive" have **<u>no spaces</u>** after them, since there are no visible characters to be printed after those letters on those lines. (For example, there are no spaces after the 'e' on that final line of output.)

As with the previous methods, if the input parameter to this method is *null*, simply print "Pshh!" to the screen (without the quotes), followed by a newline character.

(**<mark>*Important!*</mark>**) **Special Restriction:** You cannot store the output for this method in a 2D array, and you cannot create multiple 1D arrays! You must produce a single line of output (possibly using a single 1D array, if you find that useful) and print that to the screen before moving on to the next line of output. Please also refer to the special restrictions on pg. 12 before you start writing this method. Failure to abide by these restrictions will result in catastrophic point loss, even if you are passing test cases.

Parameter Restrictions: While the strings in the array passed to this method might contain space characters (' '), they will not contain any other whitespace characters (tabs, newline characters, etc.). Note that the array itself might be *null*. If the array is non-*null*, then we guarantee that none of the strings it contains will be *null*. However, some of the strings might be empty. (I.e., some of the strings might contain zero characters: "".)

Output: See above for a description of this method's output. Also, see the test cases and sample output included with this assignment for further examples of this method's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* method and therefore should not return a value.

Hints: (Highlight and/or copy and paste to reveal.)


```
public static void steamyMocha(String [] strings)
```

Description: This method exhibits all the same behaviors and restrictions as *classicRapunzel(String [] strings)*, with the exception that every string at an odd index in the array starts printing on the **second** line of output rather than the first.

For example, let's consider the following call to this method:

steamyMocha(new String[] {"clandestine", "counterintuitive", "swirlydurly", "pendulum"})

The output for that call is as follows:

```
 c     s
 l  c  w  p
 a  o  i  e
  n  u  r  n
   d  n  l  d
    e  t  y  u
    s  e  d  l
    t  r  u  u
   i  i  r  m
  n  n  l
 e  t  y
    u
    i
     t
      i
       v
       e
```

(*Important!*) **Special Restriction:** Same as *classicRapunzel()*. See also: pg. 12.

Parameter Restrictions: Same as *classicRapunzel()*.

Output: See above for a description of this method's output. Please also refer to the test cases included with this assignment. Your output for this method must match our sample output files *exactly*.

Return Value: This is a *void* method and therefore should not return a value.
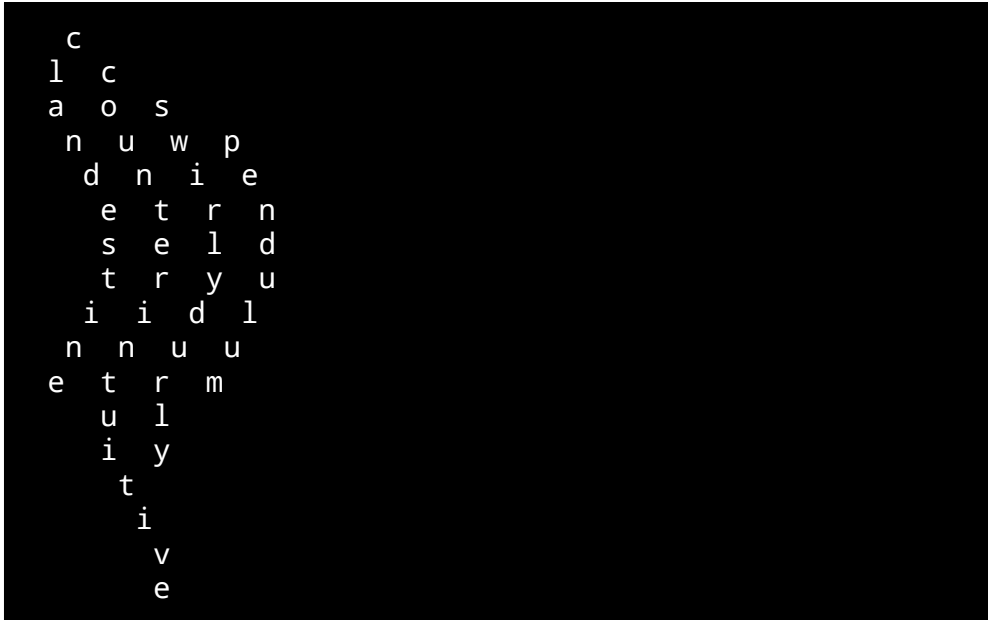
```
public static void cascadingWaterfall(String [] strings)
```

Description: This method exhibits all the same behaviors and restrictions as *classicRapunzel(String [] strings)*, with the exception that each string starts printing one line after the string before it.

For example, let's consider the following call to this method:

cascadingWaterfall(new String[] {"clandestine", "counterintuitive", "swirlydurly", "pendulum"})

The output for that call is as follows:

```
 c
l  c
a  o  s
 n  u  w  p
  d  n  i  e
   e  t  r  n
   s  e  l  d
   t  r  y  u
  i  i  d  l
 n  n  u  u
e  t  r  m
   u  l
   i  y
    t
     i
      v
      e
```

(*Important!*) Special Restriction: Same as *classicRapunzel()*. See also: pg. 12.

Parameter Restrictions: Same as *classicRapunzel()*.

Output: See above for a description of this method's output. Please also refer to the test cases included with this assignment. Your output for this method must match our sample output files *exactly*.

Return Value: This is a *void* method and therefore should not return a value.

```
public static double difficultyRating()
```

Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
public static double hoursSpent()
```

Return a realistic and reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

3.   Compiling and Running All Test Cases (and the *test-all.sh* Script!)

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. At the command line, whether you're working on your own system or on Eustis, you need to use the *cd* command to move to the directory where you have all the files for this assignment. For example:

   ```
   cd Desktop/strands_assignment
   ```

   Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type, or the entire name needs to be wrapped in double quotes. For example:

   ```
   cd strands\ assignment
   ```

   ```
   cd "strands assignment"
   ```

   It's probably easiest to just avoid file and folder names with spaces.

2. To compile your program with one of my test cases:

   ```
   javac Strands.java TestCase01.java
   ```

3. To run this test case and redirect the program's output to a text file:

   ```
   java TestCase01 > myoutput.txt
   ```

4. To compare your program's output against the sample output file I've provided for this test case:

   ```
   diff myoutput.txt sample_output/TestCase01-output.txt
   ```

   If the contents of *myoutput.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

   ```
   seansz@eustis:~$ diff myoutput.txt sample_output/TestCase01-output.txt
   seansz@eustis:~$ _
   ```

   Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

5. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *Strands.java* and all the test case files and typing:

   ```
   bash test-all.sh
   ```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting. Note that this script might have limited functionality on Mac OS systems or Windows systems that aren't using the Linux-style bash shell.

4.   Transferring Files to Eustis

When you're ready to test your project on Eustis, using MobaXTerm to transfer your files to Eustis isn't too hard, but if you want to transfer them using a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*Strands.java*, *test-all.sh*, the test case files, and the *sample_output* folder).

2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

```
scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
```

Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

5.   Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

★   Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.

★   Any time you open a curly brace, that curly brace should start on a new line.

★   Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.

★   Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.

★   Please avoid block-style comments: /* *comment* */

★   Instead, please use inline-style comments: // *comment*

★   Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"

★   The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your import statements.

★   Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.

★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.

★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.

★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do <u>not</u> have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)

★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.

★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.

★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i, j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

★ Do not use *var* to declare variables.

6.   Special Restrictions (*Super Important!*)

You must abide by the following restrictions in this assignment. Failure to abide by certain of these restrictions could result in a catastrophic loss of points.

★ (*Updated!*) For this particular assignment, the only import statement you can have is for *java.util.Arrays*. We might automatically detect assignments with *import* statements and refuse to compile them for this particular assignment, resulting in zero credit.

★ (*New!*) You cannot use any features of Java or built-in classes that we have not covered in lecture. You can, however, use any methods from any class we've covered this semester, even if we haven't covered those specific methods explicitly. (For example, we've covered the *Arrays* class already in lecture. Even though we haven't talked about every single method within that class, you are welcome to use any of its methods as you see fit.) You cannot use Java's *StringBuilder* class, even if we cover it in lecture before the assignment deadline.

★ (*New!*) You cannot modify the existing *strings* array received by any of these methods, and you cannot create any new arrays of strings. You also cannot create any 2D arrays. In each method, you can create at most a single new 1D array.

★ Your *Strands* class cannot have any member variables (i.e., fields). Every variable you create for this assignment must be declared within a method.

★ File I/O is forbidden. Please do not read or write to any files.

★ Do not write malicious code. (I would hope this would go without saying.)

★ No crazy shenanigans.

7. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *Strands.java*, via Webcourses. The source file should contain definitions for all the required methods (listed above), as well as any helper methods you've written to make them work.

Be sure to include your name, the course number, the current semester, and your NID in a header comment at the very top of your source file.

8. Grading Criteria and Miscellaneous Requirements

> *Important Note:* When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

80%   Passes test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.

20%   Adequate comments and whitespace and sound programming practices. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. For some methods, we might also check that you're using good functional decomposition and/or the DRY principle ("don't repeat yourself," also referred to in class as "never repeat the same code twice"). This portion of the grade might also award credit for including a header comment at the top of your source code with your name and NID.

Your program must be submitted via Webcourses.

Please be sure to submit your *.java* file, not a *.class* file (and certainly not a *.doc* or *.pdf* file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Important! **Programs that do not compile on Eustis will receive zero credit.** When testing your code, you should ensure that you place *Strands.java* alone in a directory with the test case files (source files, sample output files, and the input text files associated with the test cases), and no other files. That will help ensure that your *Strands.java* is not relying on external support classes that you've written in separate *.java* files but won't be including with your program submission.

9. Final Thoughts

Important! **You might want to remove *main()* and then double check that your program compiles without it before submitting.** Including a *main()* method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. Please remove.

Important! **Please do not create a java package.** Articulating a *package* in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! **Name your source file, class(es), and method(s) correctly.** Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to write any of the required methods, or failing to make them *public* and *static*, may cause test case failure. Please double check your work!

Input specifications are a contract. We promise that we will work within the confines of the problem statement when creating the test cases that we'll use for grading. For example, the *straightAndNarrow(String str, int n)* method will never receive a negative value for *n*.

However, please be aware that the test cases included with this assignment writeup are by no means comprehensive. Please be sure to create your own test cases and thoroughly test your code. Sharing test cases with other students is allowed (as long as those test cases don't include any solution code for this assignment), but you should challenge yourself to think of edge cases before reading other students' test cases.

Start early! Work hard! Ask questions! Good luck!