Programming Assignment #1: LyricalDecomposition

COP 3330, Fall 2019

Due: Wednesday, September 18, *before* 11:59 PM

Abstract

This is an introductory assignment designed to give you practice using basic Java syntax, writing Java methods, and employing some of the coding principles covered in class, such as functional decomposition and DRY ("don't repeat yourself," also referred to in class as "never write the same code twice"). By completing this assignment, you'll also get some hands-on experience with test-driven development and the testing and grading framework used in this class.

Credit for this assignment goes to Stuart Reges, who teaches computer science at the University of Washington in Seattle. This is a slightly modified version of an assignment he gives in his introductory computer programming course. Much of the verbiage used in the "Overview" section of this PDF is taken directly from his version of this assignment.

Deliverables

LyricalDecomposition.java

Note! The capitalization and spelling of your filename matter!

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

1. Overview

The goal of this assignment is to write a Java program that prints the following song lyrics using only basic *println()* statements that are appropriately arranged into static methods that don't take any arguments (with some additional restrictions listed below):

There was an old woman who swallowed a fly.
I don't know why she swallowed that fly;
Perhaps she'll die.

There was an old woman who swallowed a spider,
That wriggled and iggled and jiggled inside her!
She swallowed the spider to catch the fly.
I don't know why she swallowed that fly;
Perhaps she'll die.

There was an old woman who swallowed a bird.
How absurd, to swallow a bird!
She swallowed the bird to catch the spider;
She swallowed the spider to catch the fly.
I don't know why she swallowed that fly;
Perhaps she'll die.

There was an old woman who swallowed a cat.
Imagine that, to swallow a cat!
She swallowed the cat to catch the bird;
She swallowed the bird to catch the spider;
She swallowed the spider to catch the fly.
I don't know why she swallowed that fly;
Perhaps she'll die.

There was an old woman who swallowed a dog.
What a hog, to swallow a dog!
She swallowed the dog to catch the cat;
She swallowed the cat to catch the bird;
She swallowed the bird to catch the spider;
She swallowed the spider to catch the fly.
I don't know why she swallowed that fly;
Perhaps she'll die.

There was an old woman who swallowed a horse.
She died, of course!

You must write this program in a way that avoids code redundancy and employs good functional decomposition. In particular, you must write a single *println()* statement for each distinct line of the song. For example, the

following line appears multiple times in the song, but your source file can only contain *one println()* statement for producing this line:

> *Perhaps she'll die*

Notice that the following lines are not exactly the same, but they do have some *internal* redundancy (in that they both include substrings of the form "*She swallowed the* **X** *to catch the* **Y**"):

> *She swallowed the bird to catch the spider;*
> *She swallowed the spider to catch the fly;*

You do not have to worry about eliminating that particular kind of redundancy in your code. Since you are restricted to using simple *println()* statements, and since none of the methods you write are allowed to take input parameters, eliminating that kind of redundancy will not be possible.

In breaking this up into appropriate methods, here are some guiding principles to follow: (1) The structure of your program should capture the structure of the song. For example, there should be a different method for each verse of the song. (2) You should use clear and meaningful names for any methods you write. (3) You should avoid repeating code to the fullest extent possible. So, you should ask yourself whether or not you have repeated lines of code that could be eliminated if you were to structure your methods differently.

2. 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.

- ★ As mentioned above, each line of the song should have its own *println()* statement, and no *println()* statement should be repeated anywhere in the code.
- ★ You should employ good functional decomposition to eliminate code redundancy to the fullest extent possible. The methods you write should reflect the basic structure of the song. For example, there should be a different method for each verse in the song. You must use meaningful method names.
- ★ None of the methods you write should take input parameters. That includes your helper methods.
- ★ You cannot use any of Java's built-in classes or methods other than *System.out.println()*, and you should not use any features of Java we haven't already covered in class.
- ★ No *import* statements. We might automatically detect assignments with *import* statements and refuse to compile them for this particular assignment, resulting in zero credit.
- ★ Your *LyricalDecomposition* class cannot have any member variables (i.e., fields).
- ★ File I/O is forbidden. Please do not read or write to any files.
- ★ Do not write malicious code. (This should, of course, go without saying.)
- ★ No crazy shenanigans.

3. Method and Class Requirements

Implement the following methods in a public class named *LyricalDecomposition*. Please note that they are all ***public*** and ***static***. You may implement helper methods as you see fit (which is required for the first method, of course). 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 printLyrics()
```

> **Description:** Print the lyrics to the song given above, following all restrictions given above.
>
> **Output:** See above for a description of this method's output. Also, see the test cases and sample output included with this assignment for an example of this method's expected output. Your output must match our sample output file for this method *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.

4. 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/lyric_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 lyric\ assignment
   ```

   ```
   cd "lyric 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 LyricalDecomposition.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 *LyricalDecomposition.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.

5. 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 (*LyricalDecomposition .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!

6. 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* any 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 *not* 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.

7. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *LyricalDecomposition.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:

30%	Passes test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
30%	Follows all style restrictions. 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. Note: We expect that comments will be very sparse in this particular assignment.
30%	Uses good functional decomposition, the methods are well structured, method names are clear and reasonable, and the DRY principle ("don't repeat yourself," also referred to in class as "never repeat the same code twice") is employed effectively.
10%	Includes a header comment at the top of your source code with your name, the course number, the current semester, and your NID (*not* your UCF ID; please look up the difference if you're not sure which one is which).

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 *LyricalDecomposition.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 *LyricalDecomposition.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!

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