Assignment 4
Fun with Word Ladders
EE422C—The University of Texas at Austin

Points: **20 points.** See the Grading Details section below.

You will be working in teams of 2 for this assignment as determined in class. **No slackers allowed.** All team members will get the same initial score, which will then be adjusted by individual participation level.

The aim of this assignment is to give you experience working with recursion and various JCF sequence collections, as well as to strengthen your algorithm and ADT design skills. On this assignment I am requiring that your designs be turned in for grading – this time for quality and content. If you would like to have the early versions checked by the TA in advance, then please bring them by during his lab hours. I am also requiring that you come up with a test plan, using the techniques discussed in class.

Problem Statement:

A *word ladder*[1] is a (finite) sequence of distinct words from the English language such that any two consecutive words in the sequence differ by changing one letter at a time with the constraint that each of the resulting string of letters is a legitimate word. For example, to turn "stone" into "money", one possible word ladder is:

stone
Atone
aLone
Clone
clonS
cOons
coNns
conEs
coneY
Money

Capital letters are used in the example above only to illustrate the connections. Obviously, there could be more than one word ladder between "stone" and "money". You only have to find one of them for each word pair given.

In this assignment, you are to design and implement a Java program that for any given pair of five-lettered words, it generates **a** word ladder that connects those two words (making use of a given dictionary of legal English five-letter words). If such a word ladder does not exist between the given pair, your program should output a message that says so. Your goal is to find a (any) word ladder, not necessarily the shortest word ladder.

[1] http://www.learnenglish.org.uk/words/activities/revers01.html

Input requirements

Each line of the user input (in a file) contains exactly two words, a *starting* word and *ending* word, separated by at least one space. For example

```
smart  money
```

Each word will be in lower-cases letters. No upper case letters, and no apostrophes. You may not assume that the starting and ending words will always be legal words found in the dictionary. If either word is not in the dictionary, throw an exception and handle it accordingly by outputting an appropriate error message and then continuing with the next pair to be input. The number of input pairs is not specified and may vary, and will be provided to you in a file. The name of the file is specified in the command line. Here are a set of sample input pairs for testing your program, however you should devise additional word pairs as needed to thoroughly test your program to ensure that it does not fail or produce incorrect answers.

```
dears   fears
stone   money
money   smart
devil   angel
atlas   zebra
heart   heart
babes   child
mumbo   ghost
ryan   joe
hello buddy
hello world
heads  tails
```

Output requirements

Your program must output to the console a word ladder from the starting word to the ending word, if one exists. Every word in the ladder must be a word that appears in the dictionary. The first output word must be the starting word and the last output word must be the ending word. If there is no way to make a ladder from the start word to the end word, your program must output "There is no word ladder between ..." . If either or both input words are not legitimate 5-letter words that are in the dictionary then output a message saying so.

Sample outputs: examples

For the input words “heads” and “tails” the following word ladder was found

```
heads  heals  teals  tells   tolls   toils   tails
```

For the input words “altas” and “zebra”

```
There is no word ladder between atlas and zebra!
```

For the input words “ryan” and “joe”

```
At least one of the words ryan and joe are not legitimate 5-letter words from the dictionary
```

Solution Procedure Sketch

There are several ways to solve this problem. Your implementation must use recursion in the following manner. Create a recursive method, **MakeLadder**, which takes three parameters - the starting word, the ending word and the position that was changed to produce that word. The initial call is **MakeLadder (fromWord, toWord, 0)**. When we enter the method, we'll add the word to the **SolutionList**, i.e. the list that is displayed to the user. You will then search the **Dictionary** looking for valid words that differ from the starting word in only one letter position with that position not the same one that was just changed. All the words that match and are not already in the **SolutionList** are placed in a temporary sorted list of all candidate words. The words are prepended with the string representation of the number of letter positions that do not match the target word. This has the effect of putting words that that are closest to the solution at the beginning of the list and should lead to a solution in a shorter time. While doing this, you should check to see if the difference from the target (toWord) is one. If so, you have found a solution and can stop there. Otherwise, when the temp list is built, you will cycle through it, stripping the numeric prefixes and making recursive calls to **MakeLadder** for each entry until a solution has been found (Result is true) or you have processed the entire list. If you run out of words and no solution has been found, remove the word from **SolutionList** and exit. This will send the result of false back to the preceding level in the recursion.

Repeat this procedure until either a) you find the ending word or b) you can determine that such a word ladder between these two words does not exist. You will need to ensure that the same word does not appear more than once in a ladder, and that all words are valid words found in the dictionary. An invalid word should throw an exception that is handled gracefully by your program (i.e. no crash).

Since you will be provided with multiple pairs of start and end words, it is important to delimit one word ladder from the next. Output ********* (10 asterisks) to denote the end of the previous word ladder, and the start of the next.

Dictionary: You will find on the Blackboard in the "Assignment 4 files" folder, the file named "a4words.dat"[2], which is a text file that consists of a collection of English words with five letters each. You are to use this file as your dictionary of five-lettered words. The format of the file is that each line starts with either the character '*' or a five letter word. You can ignore the lines that start with '*'. For each of the other lines, you are to extract the five-lettered word at the beginning of the line and add that word to your internal dictionary (while ignoring the rest of the line). Store all of the words from the dictionary file in an internal **Dictionary Object** with a search capability.

Design Requirements

You should fully encapsulate the Dictionary and its search method just in case we decide to change to a more efficient data structure later. This problem is sufficiently complex as to require you to decompose, design, build, integrate and test the solution program, as a set of relatively independent parts.

Performance Output Requirement

[2] This collection was compiled by Don Knuth and is a part of the Stanford Graphbase. We are using the file as-is since it may not be modified.

You will instrument your program with the StopWatch class capability (found on the BB), and will measure and report the total run time of your program in fractional seconds.

Test Requirements
On this assignment you are to write a 1-page test plan that outlines how you will accomplish functional testing (black box), and complete branch coverage testing (white box). Show the test cases that you used to accomplish this. Submit this as a Word document along with your .java files.
Please create more than two JUnit test cases and submit your JUnit files along with your program source code.

Submission requirements
Create a package named ***assignment4*** for all of your source code.
Create a top level driver class called A4Driver.java that contains ***main ().***
Go to your project directory, and into the src directory. You will find a directory called assignment4, which in turn contains all the .java files. Put your test plan document in that directory too. Please zip the directory "assignment4" and submit it on blackboard. Do not submit individual files. Not following these instructions will be penalized.
Include a README file with team number, members' name and EID, Git repository URL, Where is your main method, and anything else you want to tell TA to help us grade your assignment.

Grading details:
Your assignment will be graded first by compiling and testing it for correctness. After that, we will read your code to determine whether all requirements were satisfied, as well as judge the overall style of your code. Points will be deducted for poor design decisions and un-commented, or unreadable code as determined by the TA. Here is the point breakdown:

Quality of the design – 3 points
Overall Correctness of the program - 10 points
Correct usage of recursion - 2 points
Proper use of exception handling – 2 points
General Design structure and Coding style - 1 points.
Test Plan – 2 points

The design of your program will require you to define and submit:

1. A System IPO diagram
2. A Use-case diagram
3. A UML model of the needed classes and their relationships
4. A functional block diagram showing the calling relationships between methods (library methods are not included)
5. The algorithm needed for the driver logic (main method)
6. A paragraph describing the rationale behind your design. This would include:
    a) How does your OOD reflect the interaction and behavior of the real-world objects that it models
    b) What alternatives did you consider? What were the advantages/disadvantages of each alternative both from a programming perspective and a **user perspective**?
    c) What are some expansions or possible flexibilities that your design offers for future enhancements?
    d) How does your design adhere to principles of good design: OOD, cohesion, coupling, info hiding, etc?

1 design per team (be sure to keep a copy of your design).

Additional Considerations

- Coding standards—You must follow the class coding standards. (See the class web page.) These standards ensure your program is readable by others. Failure to follow these standards shall result in deduction in your score.
- Understandability—Comment your program so that its logic could be explained to a layperson.
- Re-use—We may be using your program again in future assignments, so design it for future adaptations and expansion
- Efficiency Risk – It is possible that additional problem/solution constraints may be needed in order to guarantee that your program runs in a reasonable amount of time and/or space.