CMPINF 0401 -- Intermediate Programming Assignment 2
Topics: Classes, objects, data abstraction and simple files
NOT a topic: Arrays or ArrayLists

Online: Thursday, September 29, 2022
Due: All required source and text files (i.e. .java and .txt – see Issues and Hints below for specific files that must be submitted) and a completed Assignment Information Sheet zipped into a single .zip file and submitted properly (see submission guidelines) by 11:59PM on **Monday, October 17, 2022.**

Late Due Date: 11:59PM on Wednesday, October 19, 2022

Submission note: Your .zip file should contain only .java files, text files and your Assignment Information Sheet. There should be no .class files, no project files and no project subdirectories in the .zip file. Also, your TA must be able to compile your program from the command line using the javac command. Test your files to make sure this will work (especially if you use an IDE such as NetBeans for development). If the TA cannot compile your program you will receive minimal credit!

Introduction:

The Levenshtein Distance (or simply Edit Distance) between two strings is defined as follows:

Given string A of length M
Given string B of length N
Given the following operations (or edits):
Change the character at position k to a different character
Insert a character at position k (shifting the remaining characters to the right)
Delete a character at position k (shifting the remaining characters to the left)
The Levenshtein distance between string A and string B is the minimum number of edits necessary to convert string A into string B.

For example, given
A = `"PROTEIN"`
B = `"ROTTEN"`
the Levenshtein distance between A and B is 3. There may be more than one sequence of edits that achieves this minimum number. One example for the strings above is:

Delete character 0 in string A:	`ROTEIN`
Insert a 'T' in position 2 in string A:	`ROTTEIN`
Delete character 5 in string A:	`ROTTEN`

Note that the Levenshtein distance between two strings is symmetrical. If we were to start with string B in the example above, we could generate string A in 3 edits (think about a sequence of edits that would do this).

There is a famous dynamic programming algorithm to calculate the Levenshtein Distance between two strings. For more information on this algorithm see: https://en.wikipedia.org/wiki/Levenshtein_distance

The Game:

The minimum sequence of edits required to convert one string to another is sometimes obvious but sometimes it is not. It is possible to challenge a user to convert one string to another using as few edits as possible, then check to see if the user was able to achieve the minimum number.

You will implement this challenge as a simple word game. In this game two random words will be selected from a dictionary, and the user will be tasked with converting the first word into the second word in the minimum number of edits and within a short amount of time.

Scoring will be kept in two ways:

1) Out of all word pairs tried, how many words were transformed successfully within the time limit?
2) Of the words successfully transformed, how many edits were necessary vs. the minimum number required? This score will be kept as a ratio: total_edits_necessary/optimal_edits. Clearly, for this metric, a score of 1.0 is optimal and the closer the score is to 1.0 the better the performance.

A running score will be kept for each user – each time they play the game their current results will be added to their previous results. In order to store these results from run to run, they will be kept in a simple text file for each user.

Details and Requirements:

Key Helper Classes: In addition to the main program, there will be 3 key classes that you will need to use in this program. Two of them will be provided for you and the third you must write yourself.

> **Dictionary Class:** In order to implement your program you must have a way of obtaining random (actually pseudo-random) words and of determining the minimum edit distance between two strings. You must also have a way to determine if a word has been used before (see details below). The Dictionary class will do these things for you, and it has already been implemented. We discussed data abstraction in lecture, and noted that the user of a class does not need to know all of the implementation details in order to use it effectively. You will utilize this concept with the Dictionary class. It has been implemented utilizing some features / structures / algorithms which you may or may not know. However, you do not need to know its implementation in order to use it – you simply need to know the idea of what it is supposed to do and the names / parameters / general functions of its methods. See file Dictionary.java for the code for this class. Note that you really do not even have to look at the implementation details to use this class, but you should look at the comments explaining the purpose / function of each of the methods so that you know how to use it properly. Also see program DictionaryTest.java to see how the Dictionary class can be used.
>
> **MyTimer Class:** In order to limit the user to 1 minute per round, you must have some type of timer which "expires" after a minute so that you prevent further entries from the user for that round. As with the Dictionary class, the implementation details of the MyTimer class involve aspects of the Java language that may be unfamiliar to you. However, again due to the concept of data abstraction, you don't need to know these details in order to use MyTimer. See MyTimer.java and note the comments for the methods to see how they can be utilized. Because this class is a bit tricky, I have also provided a simple demonstration program to show you how to utilize MyTimer. See program TimerTest.java.
>
> **GamePlayer** class: This class you must implement yourself. It is a fairly simple class but it is very important to your game, as it will encapsulate the functionality of a player of your game. A GamePlayer will represent each player of the game and should have data to represent the player's name, number of rounds played, and other information. It should also have methods to update the values as needed and to represent the data in a form that can be saved into a file. For more specific details on the GamePlayer class and its requirements, see the program GamePlayerTest.java. Read the comments in this program carefully and see the output in GPT-out.txt. Your GamePlayer class should run with GamePlayerTest without change and should generate the same output as shown in GPT-out.txt.

Main Program: Your main program should begin by asking the user (administrator) to enter the name of the dictionary file that will be used to generate your random words. Once this is read in you should initialize a Dictionary object using this file. Note that two different dictionary files are provided – a very small one and a very large one. Note also that the Dictionary class is provided to you – so for this task you simply need to call the correct constructor.

Your main program should then allow any number of players to play the game. For each player, the program should ask for and read in the player's name. If the player's file exists (stored as "playerName.txt" for a player with name "playerName"), restore the player information from the file. Otherwise, initialize a new player. Look in the Java API for the File class to see how you can check to see if a file exists and is a regular file (i.e. not a directory). Look at the methods in the class to see which will give you the necessary information.

Once a player has been initialized, they should be able to play rounds of the game until they want to quit (or until the words have run out – see details below).

For each round of the game, you should get two random words of length between 5 and 9 (inclusive) from a Dictionary **that have not been used** during that player's current game. Note that the Dictionary class can be used both to get the random words and to check to see if the words have been used before. Look at the methods in the Dictionary class and think how it can serve both of these purposes (hint: you can have more than one Dictionary object in your program – one for the complete set of words and one for the words that have already been used). If two words that have not previously been used cannot be found, then the player should be told and the game (for that player) will end. Note: Since random words are generated, it is possible that words that were previously used are generated again by chance. This is more likely for a smaller dictionary but it is possible for any size dictionary. Thus, before declaring that no more unused words can be found, you should try several times to generate the words.

For each round of the game the player will have 1 minute to transform the first random word into the second, using the edits described above. Specifically, a user will be allowed to do the following:

- **`C k v`** – This command will **`C`**hange the character at location **`k`** to new value **`v`**
- **`I k v`** – This command will **`I`**nsert at location **`k`** with value **`v`**
- **`D k`** – This command will **`D`**elete the character at location **`k`**

In the example above, to convert "PROTEIN" to "ROTTEN", the user would type

```
D 0
I 2 T
D 5
```

After each edit you should show the user the current word and the goal word in a nicely formatted way, so they can see what they still have to do to complete the transformation. A **StringBuilder** can be used for the current word and **all of the edits can be done via StringBuilder mutators**. See course handouts and the Java API for StringBuilder to see which methods will be useful for your edits. See a2out-1.txt and a2out-2.txt for examples of how this process would proceed.

A round ends when the user either completes the transformation or runs out of time. If the user completes the transformation, the success() method should be called from the Player object with the appropriate parameters. If the user runs out of time the failure() method should be called from the Player object. The idea here is that after a round completes you are mutating the Player object to reflect the success or failure of that round. **For more details on the success() and failure() methods see the GamePlayerTest.java program.**

When the user elects to quit or runs out of unused words, the game ends for that player. At that point the player info should be saved back to the file (under the player's name). See GamePlayerTest.java for an idea of how to do this.

When a player's game ends, the main program should prompt for another player to play the game. This could be any player (including the player that just finished playing). If a player who has already played the game plays again, the set of used words should be reset – and they can be used again.

If no more players want to play the game then the program should end.

If you want a sample of how your output should look, see my output in a2out-1.txt and a2out-2.txt. Your program does not have to look exactly like this, but the functionality should be the same and the formatting should be good (so a user would actually enjoy playing the game). Note that in a2out-1.txt a large dictionary file is used, while in a2out-2.txt a small dictionary file is used – and you can see how the game ends for a player when they run out of words to try.

Important Note 1: You should **NOT use any array, ArrayList or any other predefined Java collections** of data (other than String and StringBuilder and the provided Dictionary class) in this program. All of the data structures you need are provided in the 3 helper classes. One of the goals of this project is for you to utilize someone else's classes effectively within your program.

Important Note 2: There is a lot of functionality in your main program. You should definitely think of dividing this up using methods.

Issues and Hints:

- Test your program thoroughly before handing it in. Make sure it functions as required in all cases.
- Don't forget your **Assignment Information Sheet --** you will lose points without it.
- Don't forget to **comment your code** – you will lose points without them.
- Note that you must have **8 files** in your .zip submission file for this assignment:
    1) Assig2.java [written by you]
    2) GamePlayer.java [written by you]
    3) GamePlayerTest.java [provided]
    4) Dictionary.java [provided]
    5) MyTimer.java [provided]
    6) dictionary.txt [provided]
    7) smallDict.txt [provided]
    8) Your assignment information sheet

  Make sure all of these files are present or you will lose submission points!

Extra Credit Option (up to extra 10 points total):

- Add a password to the user's file and require each user to enter a password in order to play the game. If a user cannot remember the password they will have to pick a different name in order to play the game **[5 points]**
- [Very challenging – much more difficult than the 10 points you will get for it] Have your program determine the optimal edits that are used to convert one word to the other and show the optimal sequence of edits at the end of each round. Note that in order to do this you need to understand how the Levenshtein Distance is calculated, and how the actual moves can be derived, which may require some research. I don't recommend this but if you finish the program very early and are looking to do something where the cost will be much more than the benefit, you can try this. **[10 points]**