CS 251 Intermediate Programming
Project 7: Word Search Solver

Brooke Chenoweth

Spring 2025

Program Description

For this project, you will create a computer program that can solve word search puzzles.

Your main class (the one with the main method of the program) should be named "WordSearch". It is possible, though very much *not* recommended, to write your entire program inside of `WordSearch.java`. I expect most of you will break up the program into multiple classes and may have additional main methods used for testing those classes. This is fine, but make sure the actual main method for us to use to run your program is in the `WordSearch.java` file.

Command Line Arguments

Your program will take three command line arguments.

1. The first argument is the name of the file containing the words your program may find in the puzzle.

   Each word will be on a separate line of the file. Make no assumptions about the order of the words. (That is, we may test your program with a word list that is not alphabetized.)

2. The second argument is the name of the file containing the puzzle itself.

   The first line of the file contains two numbers separated by a space. The first number is the number of rows in the puzzle and the second is the number of columns in the puzzle.

   The remaining lines in the file are the rows of word search puzzle, with no spaces between the letters.

3. The third argument specifies which directions to search for the words. You must recognize at least the following arguments:

   - ALL – All eight possible directions, horizontal, vertical, both diagonals, going forwards and backwards.

- HORIZVERT – The four horizontal and vertical directions, forwards and backwards. (Everything but the diagonals.)
- DIAGONAL – The four diagonal directions, forwards and backwards. (Only the diagonals)
- FORWARD – The four "forward" directions. If we use compass directions with north being at the top of the puzzle, these would be northeast, east, southeast, and south.

If you wish, you may specify additional directions in your command line arguments beyond the four described here. If you do so, please document it in your Javadoc comment on the main method of the WordSearch class.

Program Output

Your program will print all the words from the dictionary found in the puzzle, checking all specified directions from each spot. It is possible for words to overlap so if, for example, the puzzle contains the word "catalog", your solution will also include "cat", "at", and "log", as well as "catalog", assuming those words are all in the word list file.

Each line of output will consist of a word found in the puzzle, the row and column where the word started, and the direction from that starting position where the word continues, each separated by spaces.

- When specifying row numbers, the top row of the puzzle is row 0 with row numbers increasing as you move down.
- When specifying columns, the leftmost column is column 0 with column numbers increasing as you move right.
- When specifying directions, we'll use compass directions, so N is up, E is to the right, SW is diagonally down and left, etc.

It is possible that the same word may appear multiple times in the puzzle and, indeed, may even appear twice starting from the same position, so order the lines of output as follows:

- If the words differ, output them in lexicographic order. (That is, just use the natural comparison from the String class to order them.)
- If the words are the same, compare the row numbers and output the line with lower number first.
- If words and row numbers are the same, compare the column numbers and output the line with the lower number first.
- If word, row, and column are the same, compare the direction and order them according to the following: N, NE, E, SE, S, SW, W, NW

These puzzles have very large expected output which would be infeasible to verify manually when grading, so your output must match the sample output exactly, including order of the output lines, whitespace, capitalization, etc.

Testing Your Program

I am providing you with several word list files and several puzzle files along with their expected output for testing. The graders will test your code with some of these files as well as some unknown test files.

As I said before, the output for all but the smallest of puzzles is to much to verify by eye, so you should output redirection to capture your programs standard output in a file and then use a utility like `diff` to compare it with the expected output. (Alternatively, you can use the `M-x diff` command in emacs to compare them, or some other text comparison program.)

Example

If the word list is in a file named `words.txt`, the puzzle is in `puzz.txt`, and the expected output is in `expected.txt`, you can test your program with the following commands.[1]

```
java WordSearch words.txt puzz.txt ALL > myOut.txt
diff myOut.txt expected.txt
```

The first command runs your program and redirects it to a file named `myOut.txt`. The second command uses the diff command to perform a character-by-character comparison of your output with the expected output. If diff returns silently (without printing anything), then you have passed all tests. If it does print something, you'll be able to see which lines differ. Punctuation, capitalization, and whitespace all matter here![2]

Turning in your assignment

Once you are done with your assignment, use Canvas to turn in `WordSearch.java` and any other java files you may have created for this program. Note that each source code file should still contain a header comment with your name, class, and other related information. Failure to properly comment your files may lead to point deductions. *Please make sure to use proper Javadoc comments in all your files.*

[1] I'm assuming you already compiled your code and that you are on a system with the diff utility.

[2] If generate your output on a Windows system, you may find the output differs because of the line termination characters. (Graders will not count off for this.) In this case, you may want to either tell your text comparison program to ignore the whitespace difference at the end of the line, or resave the files in the same format.