import java.util.Arrays;

import java.util.Scanner;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.PrintWriter;

/\*\*

@author Wasfi Momen wmomen1@student.gsu.edu

Purpose: The purpose of this program is to take a file with words and see if any of them are anagrams. If there are we will print them out on a single line in the output file. The objective is to gain experience in String objects, arrays, and generic algorithms as well as experiment with Java's File I/O methods.

Solution: The solution for this program will open the file, sanitize the data so that we can compare strings without any punctuation or case issues, compare the strings using the characters and length of the String to see if it is an anagram, and then outputting the original Strings to another file.

Data Structures Used: Arrays

Program Use: User will input a file containing words, program computes the anagram of words, and outputs a file output.txt with all the words that are anagrams of each other on the same line.

\*/

public class anagram {

// inHere() checks to see if whether the int array containing the indexes of the

// of the sorted array of Strings containing the signatures acutally contains a particular value.

// We use inHere() to check for whether a value in the signatures array has already been scanned and identified as

// a set of anagrams. That way, we handle the problem of printing out a word on a new line even though it has already been

// identified as anagram and is already on another line.

//

// Precondition:

// - Array1 has already been sanitized without any punctuation and case. It also assumes array1 has no null values.

public static boolean inHere (int [] array1, int value) {

boolean isHere = true;

for(int m = 0; m < array1.length; m++) {

if (value == array1[m]) {

return true;

}

}

return false;

}

// fileIO() contains the main logic for the program by doing the file handling, creating three arrays for each step

// in the process of taking words from the input file, sanitizing the words, and outputting based on logic from other methods.

// Generally, this method could be split into other methods for easier readability, but I have added comments whereever necessary to

// explain the process.

// fileIO fits the requirements for the <50 word limit, <=12 character per word limit, and no file exceptions.

public static void fileIO () {

// Provide keyboard input to give file name

Scanner keyboard = new Scanner(System.in);

System.out.println("Please provide the path of the file: ");

String userFile = keyboard.nextLine();

try {

// Create new File object to take in file.

// Make new Scanner to take in file for processing.

File file = new File(userFile);

Scanner inputFile = new Scanner(file);

int counter = 0; // To count for how many tokenized words are in the file, if >50 abort program.

String input = ""; // For unsanitized String input from the file.

String sanitize = ""; // Processed input that will not have any punctuation or case.

// Since there can only be 50 words in the file before the program breaks,

// we can declare a String array to hold our String input from the File.

// List would be preferred, but we are using Arrays to solve this problem.

String [] original = new String [50]; // Original Strings in the File.

String [] processed = new String[50]; // Processed Strings to be sorted.

while(inputFile.hasNext()){

input = inputFile.next();

if (counter < 50){ // Check to see if inputFile contains more than 50 words.

// Check to see if the string is more than 12 characters long and process if it is.

if (input.toCharArray().length < 12) {

// Create a new String sanitize to take in input and remove punctuation and case.

// The regex pattern ("[^a-z\\s]", "") removes anything not matching a-z or whitespace.

original[counter] = input;

sanitize = input.toLowerCase().replaceAll("[^a-z\\s]", "");

processed[counter] = sanitize;

counter++;

}

} else {

// Exit if inputFile contains more than 50 words.

inputFile.close();

System.out.println("Input file contained more than 50 words.");

System.exit(1);

}

}

// Close the file.

inputFile.close();

// Calculate a new array size based on how many of the values are null

// in our 50 word array.

int newArrSize = 0;

for (int s = 0; s < processed.length; s++){

if(processed[s] != null) {

newArrSize++;

}

}

// Processed1 is a copy of Array processed without

// null values. This makes it easier to print out without errors.

String [] processed1 = new String [newArrSize];

int index = 0;

for (int b = 0; b < processed.length; b++) {

if(processed[b] != null) {

processed1[index] = processed[b];

index++;

}

}

// Original1 is a copy of Array original

// without null values. This makes it to print easier without errors.

String [] original1 = new String [newArrSize];

int index1 = 0;

for (int c = 0; c < original.length; c++) {

if (original[c] != null) {

original1[index1] = original[c];

index1++;

}

}

// Create a new Array signatures to hold the char-sorted

// Strings from processed1.

String [] signatures = new String [newArrSize];

String sorted = "";

for(int l = 0; l < processed1.length; l++) {

char [] firstWord = processed1[l].toCharArray(); // Convert each word into a char array.

Arrays.sort(firstWord); // Sort the char array to get an alphabetical String as a signature.

sorted = new String(firstWord); // We can use JVM's inherit ability to use a char array to build a new String.

signatures[l] = sorted; // Fill the signatures array with the char-sorted Strings.

}

// beenSorted is the array that will input into inHere().

// It contains the index values matching the position of the signature array as a one-to-one relation.

int [] beenSorted = new int [newArrSize];

Arrays.fill(beenSorted, newArrSize+1); // We fill the beenSorted array intially with all false values.

PrintWriter output = new PrintWriter ("output.txt"); // PrintWriter handles file IO.

// The outer loop will go through the signatures array to print the first word in the array only if

// it is not already in the beenSorted array.

// The inner loop finds the anagrams throughout the signatures array by looking for equivalent String values and neglecting any

// indexes from the signatures array that have already gone through the loop and printed the anagrams to one line.

for(int v = 0; v < signatures.length; v++) {

if (!inHere(beenSorted, v)) {

output.print(original[v] + " ");

}

for (int c = 0; c < signatures.length; c++) {

if( (v != c) && (signatures[v].equals(signatures[c])) && (!inHere(beenSorted, c)) ) {

output.print(original[c] + " ");

beenSorted[c] = c; // Add the outer loop index value to the beenSorted array since we have found its anagram(s).

beenSorted[v] = v; // Add the inner loop index value(s) to the beenSorted array since we have found its anagrams(s).

}

}

output.println();

}

// Close the PrintWriter.

output.close();

} catch (FileNotFoundException e){

// Catch the exception for if there is no file and print out message.

System.out.println("The input file is empty.");

System.exit(1);

}

}

public static void main (String [] args) {

fileIO();

}

}