Lab 2 in TND002

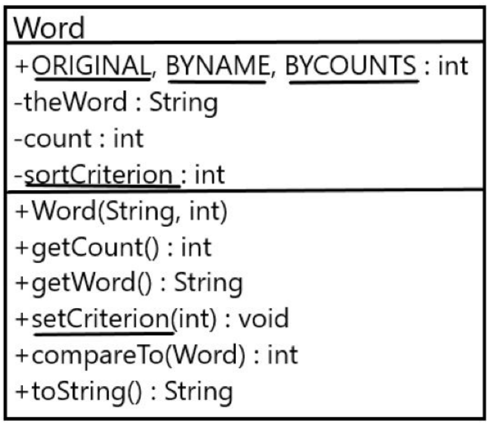
Mark Eric Dieckmann

November 8, 2023

1 Summary

The aim of this lab is to familiarize you with how Java reads from and writes data to the console and hard disk. You will also handle exceptions and use try-catch blocks. You will read in text from an external file, break it up into words, store the words in a dynamic array that serves as a dictionary and sort them using various criteria. You will input text from the console and save data to an external file. You implement three classes. **Word** contains variables and methods that are relevant for individual words of your text. **Dictionary** stores all words and provides methods that are applied to the ensemble of words. **Lab2** contains *main(..)* and handles all the reading and writing from and to the console and external files.

2 Task1: The class Word



Word has the class diagram shown to the left. Set the class constants to the values 0, 1, and 2 when you declare them. Use the name of the constants in your code (in all classes) instead of the values. This makes the code easier to read. Initialize *sortCriterion* with *ORIGINAL* when you declare it.

The instance variables *theWord* and *count* are initialized in the constructor.

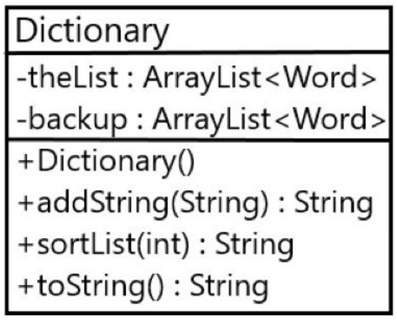
getCount() and *getWord()* return the values of *count* and *theWord*.

setCriterion(arg) changes the value of *sortCriterion* to one of the class constants.

compareTo(arg) should always return 2 if *sortCriterion* == *ORIGINAL*. Otherwise, it compares two instances of **Word** either by the values of *theWord* or by those of *count*. Which one, depends on the value of *sortCriterion*. If *sortCriterion* equals *BYNAME*, *compareTo(arg)* should compare the values of *theWord* alphabetically. Use the instance method *compareTo(..)* of **String** with the same name as yours (which compares two instances of **Word**) to compare the values of both strings *theWord* alphabetically. If *sortCriterion* equals *BYCOUNTS*, then your *compareTo(..)* method should compare the values of *count* numerically. For these two criteria, your *compareTo(arg)* method should send back one of the possible values -1, 0, or 1. It should return a value -1 if the value of the instance variable (either *count* or *theWord*) of the calling instance of **Word** is smaller than that of *arg* in the argument list of *compareTo(arg)*. It should return 0 if the values of the instance variables match. If the value of the instance variable of the calling instance is larger than that of its counterpart in *arg*, *compareTo(arg)* should return 1.

toString() returns a formatted string. It starts with "Word:" followed by the value of *word* in a column 10 characters wide and aligned to the right. You leave 3 empty spaces and write "Count:" followed by the value of *count* in a column 3 characters wide.

3 Task 2: The class Dictionary


Dictionary
-theList : ArrayList<Word>
-backup : ArrayList<Word>
+Dictionary()
+addString(String) : String
+sortList(int) : String
+toString() : String

The instance variables of this class are *theList* and *backup*. You initialize *backup* to *null* when you declare it. *theList* is initialized in the constructor.

addString(arg) takes in the string *arg*. If *arg* is not yet contained in any element of *theList*, then *addString(arg)* creates a new instance of **Word** with a value *count* = 1 and adds it to *theList*.

If there is already an instance of **Word** with a value of *theWord* equal to *arg* in *theList*, *addString(arg)* increases its value of *count* by 1. You can accomplish this by replacing this instance of **Word** by a new instance with a value of *count* that is one higher than the previous one. *addString(String)* should return the return value of *toString()* of the added or updated instance of **Word**.

sortList(arg) sorts the instances of **Word** in *theList* according to the value of *arg*. If *sortList(arg)* is called for the first time (*backup* is *null*), then it should attach a deep copy of *theList* to *backup*. If *arg* of *sortList(arg)* has the value of the class constant *ORIGINAL*, then it should (shallow) copy the address of *backup* to *theList*, set the value of *sortCriterion* to that of *ORIGINAL*, and return "Word list was reset".

If *arg* has the value of one of the other class constants, then you set *sortCriterion* in **Word** to *arg* and you sort the elements of *theList*. Use the method *compareTo(arg)* of **Word** to compare the instance of **Word** at the current position in *theList* to the instances in the slots with a higher index. You implement a nested loop. The outer loop (loop index i) goes from the first to the second-last element of *theList*. The inner loop over j goes from the index (i+1) until the last element of *theList*. If the result of *compareTo(arg)* applied to the words selected by the outer loop (element i) and the inner loop (element j) is -1, you swap both instances of **Word**. In the end, you get a list that is either sorted by the number of *count* or alphabetically by the value of *theWord*.

The elements of *theList* will be sorted in descending order. *sortList(arg)* should return ”Sorted by counts” or ”Sorted alphabetically” depending on the value of *sortCriterion.*

If the value of *sortCriterion* does not correspond to any of the class constants of **Word**, then *sortList(arg)* should not do anything and return “Sort criterion not known”.

toString() should return a string that starts with ”Content: ” followed by a line break. It should call the *toString()* methods of all elements of *theList* and concatenate their return strings to one large string. There should be a line break between each. The method should return the concatenated string.

4 Task 3: The class Lab2

The method *main(..)* of **Lab2** deals with IO from the console and from/to external files. You throw the IOExceptions from the console and you catch IOExceptions from the file access (to practice both). Know the difference between both schemes,

Initialize a global console reader in **Lab2** and create in *main(..)* an instance of **Dictionary** called *theDictionary.*

Write ”Type filename:” to the console and read in a string from the console. Initialize with it an instance of **File** and check if the file exists. If it does not exist, read another filename from the console and try again. If it finds the file, then you should read all the lines of the file and create one long string with it. Trim it using the instance method *trim()* of **String** to avoid getting empty characters at the beginning or end. Change all letters of the long string to lowercase ones.

You split the string at one or more empty spaces, creating a static (= fixed size) array of individual strings. You send all strings of the static array, which are not numbers, into the *addString(arg)* method (one by one) of *theDictionary* and write the return value of this method to the console. This helps you keeping track if the value of *count* increases as it should when you send in a word that already exists. The output of this step is shown in Fig. 1.

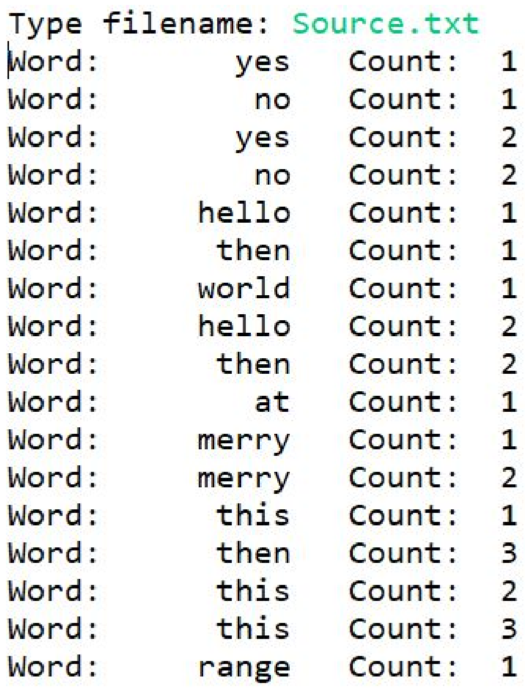
Write the return value of *toString()* of *theDictionary* to the console. See Fig. 2.

Sort the list in *theDictionary* by counts and write the return string of *sort(..)* to the console. Write the return value of *toString()* of *theDictionary* to the console (See Fig. 3).

Sort the list in *theDictionary* alphabetically and write the return string of *sort(..)* to the console. Write the return value of *toString()* of *theDictionary* to the console (See Fig. 4).

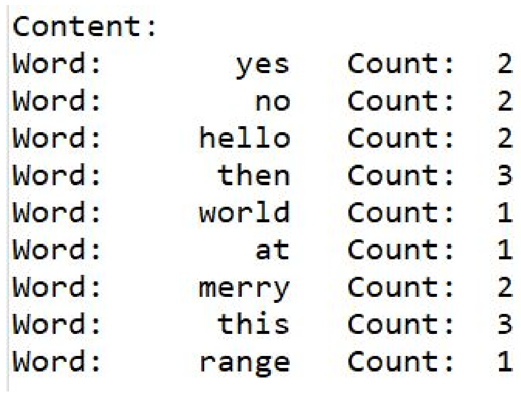
Restore the original list and write the return value of *sort(..)* to the console. Write the return value of *toString()* of *theDictionary* to the console (See Fig. 5).

Create a file called ”result.txt” and you write the return value of the *toString()* method of **Dictionary** into it. The content in the file “result.txt” should be identical to Fig. 5. Close all readers and writers when you are done.



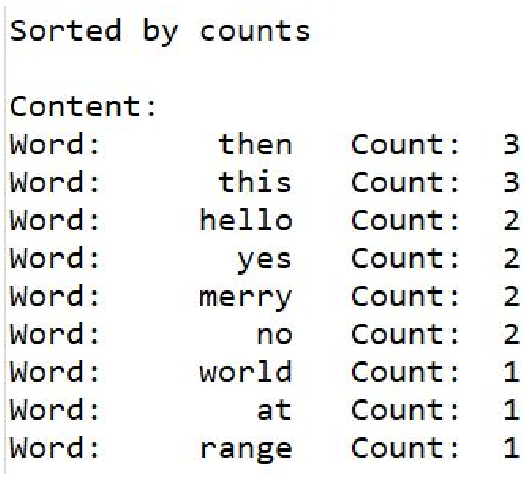
```
Type filename: Source.txt
Word:        yes   Count:  1
Word:         no   Count:  1
Word:        yes   Count:  2
Word:         no   Count:  2
Word:      hello   Count:  1
Word:       then   Count:  1
Word:      world   Count:  1
Word:      hello   Count:  2
Word:       then   Count:  2
Word:         at   Count:  1
Word:      merry   Count:  1
Word:      merry   Count:  2
Word:       this   Count:  1
Word:       then   Count:  3
Word:       this   Count:  2
Word:       this   Count:  3
Word:      range   Count:  1
```

Figure 1: The console output while adding words to Dictionary



```
Content:
Word:        yes   Count:  2
Word:         no   Count:  2
Word:      hello   Count:  2
Word:       then   Count:  3
Word:      world   Count:  1
Word:         at   Count:  1
Word:      merry   Count:  2
Word:       this   Count:  3
Word:      range   Count:  1
```

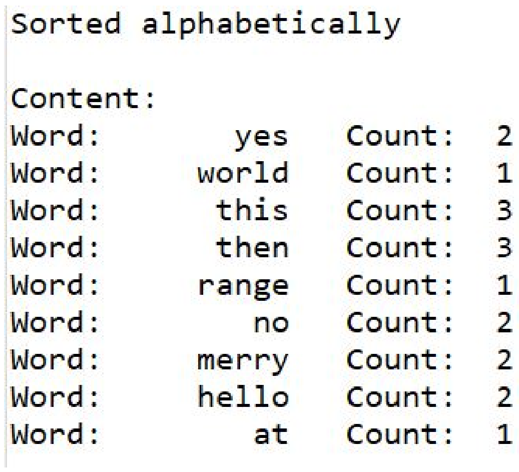
Figure 2: The console output of Dictionary.toString() after setting up the array list



```
Sorted by counts

Content:
Word:       then   Count:  3
Word:       this   Count:  3
Word:      hello   Count:  2
Word:        yes   Count:  2
Word:      merry   Count:  2
Word:         no   Count:  2
Word:      world   Count:  1
Word:         at   Count:  1
Word:      range   Count:  1
```

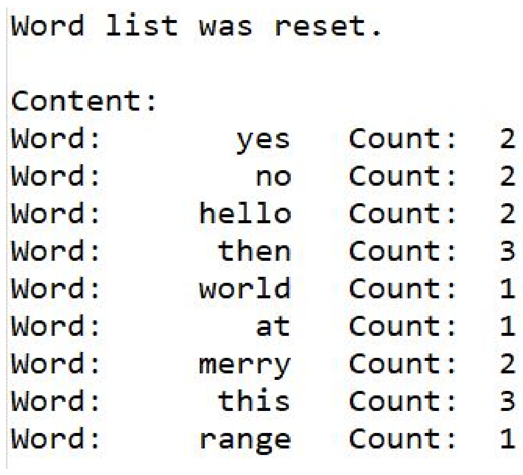
Figure 3: The console output of Dictionary.toString() after sorting by counts



```
Sorted alphabetically

Content:
Word:        yes   Count:  2
Word:      world   Count:  1
Word:       this   Count:  3
Word:       then   Count:  3
Word:      range   Count:  1
Word:         no   Count:  2
Word:      merry   Count:  2
Word:      hello   Count:  2
Word:         at   Count:  1
```

Figure 4: The console output of Dictionary.toString() after sorting alphabetically



```
Word list was reset.

Content:
Word:       yes   Count:  2
Word:        no   Count:  2
Word:     hello   Count:  2
Word:      then   Count:  3
Word:     world   Count:  1
Word:        at   Count:  1
Word:     merry   Count:  2
Word:      this   Count:  3
Word:     range   Count:  1
```

Figure 5: The console output of Dictionary.toString() after restoring the list