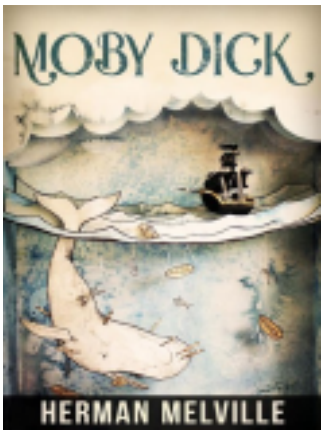
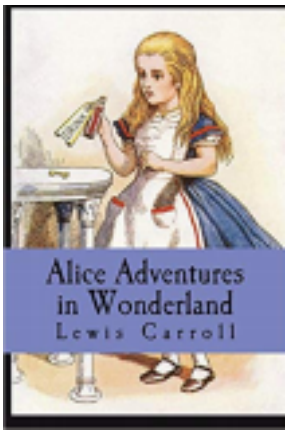
CSC220 Lab13
WordFisher

The goal of this project is to:

1. Practice working with the data structures we have learned this semester.
2. Practice I/O.
3. Use Java APIs in a software project.





Digital humanists work with all kinds of computational tools when studying and analyzing texts. Having learned and implemented lots of data structures and algorithms this semester, we can build our own text analysis program called **WordFisher** to answer questions these researchers might be interested in asking. Here are some questions our tool should be able to address:

1. What is the **total number of words** in a text?
2. Given a word, **how many times does it occur** in the text?
3. What are the top **10 most frequently occurring words** in a text?
4. What are the **most common popular words *between* two texts**?

To make our tool more robust, WordFisher should also have an option to filter out the most commonly used words in the language when reporting the most popular words in the text, e.g. “the”, “him”, “you”, “but”, “I”. These are usually called *stopwords*. Furthermore, we will do some basic preprocessing of the text: counting lower- and upper-case versions of a word as the same word, e.g. there is no difference between “Sea” and “sea,” and stipulating that all words must be alphanumeric, e.g. “?” or “--” is not allowed.

We will use two texts for our testbed: **(1) Herman Melville’s *Moby Dick*** and **(2) Lewis Carroll’s *Alice in Wonderland***. The plaintext version of these are provided for you.

To solve this problem, we will use implementations from the Java API to help reduce the work needed (i.e. we won’t reinvent the wheel by implementing data structures on our own) and make our code as efficient as possible. One interface we will use is **Map** (as you learned in lecture), which *maps* a key to a value. This is particularly useful because we can let key be a word in the text, and value its frequency. For instance, if our Map is called vocabulary, we can say:

```
vocabulary.get(“handkerchief")
```

which would return the number of times the word *handkerchief* occurs in the text. Map, however, is only an interface. Two implementations of it are **HashMap** (hash table based) and **TreeMap** (based on binary-search trees that are balanced). Because we want our looks-ups to be as fast as possible and our collection of texts are not too large in size, we can use **HashMap**. What would be the O() of our lookups then? What if we used TreeMap instead?

Part 0

- Create a new Java project and call it **Lab13**.
- Create a package inside your project and call it **lab13.**
- Download the **Lab13** ZIP folder from BB (Google Drive Link) and copy in the following files **into the PROJECT (not the package)**:
    - **texts/** This directory contains the input texts you have been asked to work with, available in plaintext. Note that the words are **space delimited**.
    - **stopwords.txt** This file contains a list of stopwords, **one word per line**.
- Copy **WordFisher.java** and **WordFisherTester.java** into your new lab13 package.

Part 1 — What to Do

We will develop the functionality of this program in steps, each method building atop the other. **Your WordFisher class is required to have the following methods with the following signature and nothing else. The grader will penalize accordingly if he finds anything else!** However, as always, you are encouraged to write as many helper methods as you need. It is also advised that you do your implementation in the order given; it will make your job easier!

1a. `public WordFisher(String inputTextFile, String stopwordsFile)`

- This constructor receives an input file name and a second file name containing stopwords. As usual, all member variables of the WordFisher class need to be initialized. The member variables of this class are **(copy them exactly!)**:
    - `public HashMap<String, Integer> vocabulary`
    - `private List<String> stopwords` (to be instantiated as an ArrayList)
    - `private String inputTextFile`
    - `private String stopwordsFile`
- To initialize vocabulary and stopwords, the constructor should call a method called `getStopwords()` and another called `buildVocabulary()`.

1b. `private void getStopwords()`

- This method populates the stopwords list from a file containing all

~~stopwords, as pointed to by the member variable stopwordsFile.~~ ~~**This file contains one stopword per line.**~~ ~~You should be familiar with how to read from a file from the previous labs.~~

1c. `private void buildVocabulary()`

- This method populates the map vocabulary from a file containing the full text in plaintext format. **Note that each word of the text is space delimited.** Additionally, the text may contain non-alphanumeric characters like “?”, “-- ”, and “)” which must be filtered out.
- Here is a way you can read in the text as a String[] using Java Files and Paths:

```
String reader = new String(Files.readAllBytes(Paths.get(fileName)));
String[] allWords = reader.split("\\s+"); // any # of spaces
```

    - (
    - Be sure to include the following imports:
    - import java.nio.file.Files;
    - import java.nio.file.Paths;
    -)
- To **filter the text for non-alphanumeric characters and ensure that all words are made lowercase**, you should amend the above lines to call **`toLowerCase()`** and **`replaceAll("[^a-zA-Z0-9 ]", “”)`**. The expression [^a-zA-Z0-9] is a regular expression that asks to replace everything *except* those characters that are a through Z, A through Z, 0 through 9, and whitespace.
- When visiting the elements in allWords, if a word is not yet seen in the map, a new entry must be made for it (with what frequency?). Otherwise, if the word already exists in the map, get the current frequency and update the entry with the new frequency.
- You should refer to the HashMap documentation for help as to what methods the Map interface offers for adding, updating, removing, getting, and checking if a key exists.

2. `public int getWordCount()`

- This method returns the total number of words in the text and can be obtained using the map `vocabulary`.
- The total number of words in Moby Dick is **218,619**. Alice has **27,336**.

3. `public int getNumUniqueWords()`

- This method returns the total number of *unique* words in the text. This can be obtained using the map vocabulary.
- The total number of unique words in Moby Dick is **17,139** and Alice **2,570**.

4. `public int getFrequency(String word)`

- This returns the word frequency for a given word. This can be obtained using the map vocabulary. If the word does not exist in the vocabulary, -1 should be returned.
- For instance, the word “whale” occurs 1,226 times in Moby Dick! “handkerchief” occurs 5 times in Moby Dick and does not occur in Alice (thus, returns -1).

5. `public void pruneVocabulary()`

- This method removes all stopwords from vocabulary.
- After pruning, getWordCount() on Moby Dick returns 110,717 words; Alice returns 12,241. (that’s a lot of words removed!)

6. `public ArrayList<String> getTopWords(int n)`

- This method receives an integer n and returns the top *n* most frequently occurring words in the text as an **ArrayList** of strings.
- To implement this, you will need to use the **PriorityQueue** offered by Java (what’s an implementation of Priority Queue we learned?). The elements of this PriorityQueue should be a custom object, call it **WordNode**, that stores two fields, a word and its frequency. This can be a nested class (a class defined within another class), as we saw in the Pacman lab (Lab12).
- We also need to be able to compare, or order, two WordNode objects (why?). Therefore, we must create a class (you can name it whatever) that implements **Comparator**. We learned how to do this in Lab04. What should these two WordNode objects be compared by?
- The PriorityQueue should be instantiated with an instance of this Comparator class you made. This PriorityQueue should then be offered all words from vocabulary as WordNode objects.
- With this PriorityQueue now in place, how can we extract the top n most frequent words? Here is an easier question: where is the most frequent word in this data structure? Think about it before writing code…

- When calling **getTopWords(10)** on the pruned vocabulary of **Moby Dick**, the following list is returned: (what would it return if it was unpruned?)
    - **[whale, one, like, upon, man, ship, ahab, ye, sea, old]**
    - We learn that whales, men, and ships are particularly important in this story :-) What does Alice give you?
    - To verify the results, you may wish to write a helper method that takes this resulting list as input and prints the associated frequency. The frequencies should appear in descending order.

7. `public ArrayList<String> commonPopularWords(int n, WordFisher other)`

- This method receives an integer ***n*** and another **WordFisher** object (i.e. another text) as input and returns an ArrayList of the common popular words (taken from the *n* most popular from the first text, *n* most popular from the other) between the two texts. An empty list should be returned if there are no common words.
- For instance, calling this method on the pruned Moby Dick with pruned Alice and *n* = 20 gives… [one, like, would, time]

Part 2 — Final Remarks

This lab is worth double the points of a regular assignment. Do it well!

- You are fully in charge of the implementation. Use the notes and hints above to help you.
- Test your program thoroughly using the testing techniques we have learned in the semester.
- Your WordFisher class must contain **ALL** of the above listed methods with the exact same signatures as listed.
- You are welcome to use as many helper methods as you need :-)
- **Start your assignment early, and seek help early (either from the instructor, the LA, TAs, etc.).**

Make sure to upload your Lab13 folder to Box when you have completed this assignment by FRIDAY May 7th at 11:59pm.