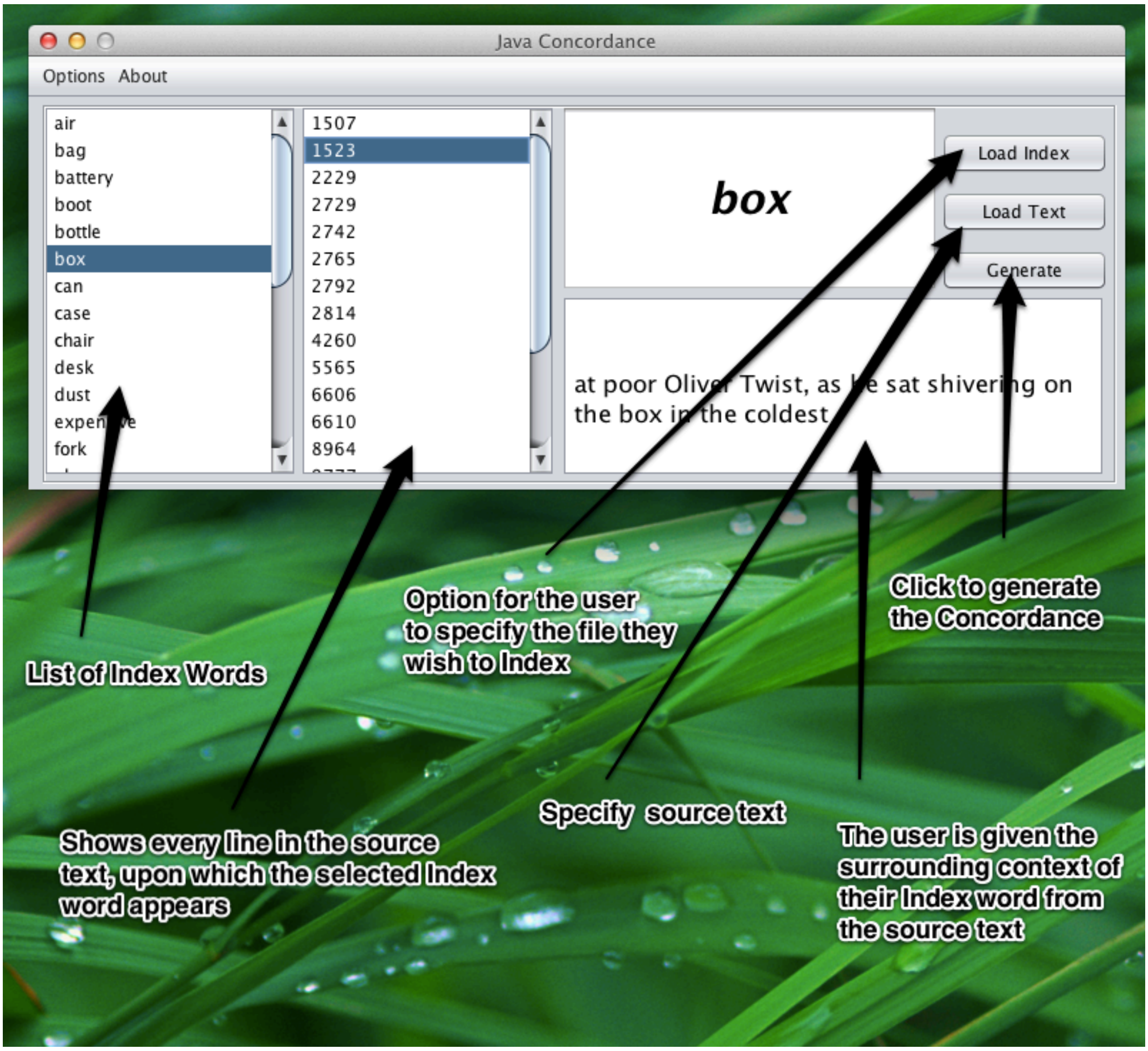
Java Concordance Assignment

The purpose of this program is to allow a user to generate a Concordance of a source text file and return the line numbers for each reference to an index. Context is also provided to give the user more information about their chosen index word.

My implementation of the solution to this problem evolves around a Graphical User Interface, to allow the user to select their index and source text files. The user can then generate a Concordance by clicking "Generate" which will call two algorithms to process and display the Concordance.

Upon starting the application, the user is greeted with a pop up explaining how to use the application however it is not difficult to use.


Java Concordance
Options
About
box
Load Index
Load Text
Generate
List of Index Words
Shows every line in the source text, upon which the selected Index word appears
Option for the user to specify the file they wish to Index
Specify source text
Click to generate the Concordance
The user is given the surrounding context of their Index word from the source text

air
bag
battery
boot
bottle
box
can
case
chair
desk
dust
expensive
fork

1507
1523
2229
2729
2742
2765
2792
2814
4260
5565
6606
6610
8964

at poor Oliver Twist, as he sat shivering on the box in the coldest

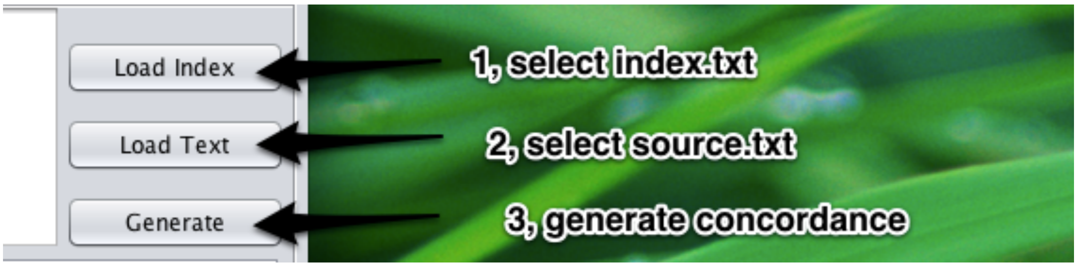
The interface has been designed to be intuitive and should be useable by anyone that has used any sort of program before.

Two scrollable JLists are used to display both index words and line numbers, the context is then written to a JTextArea and 3 JButtons are available to provide the two files and to generate the Concordance itself.

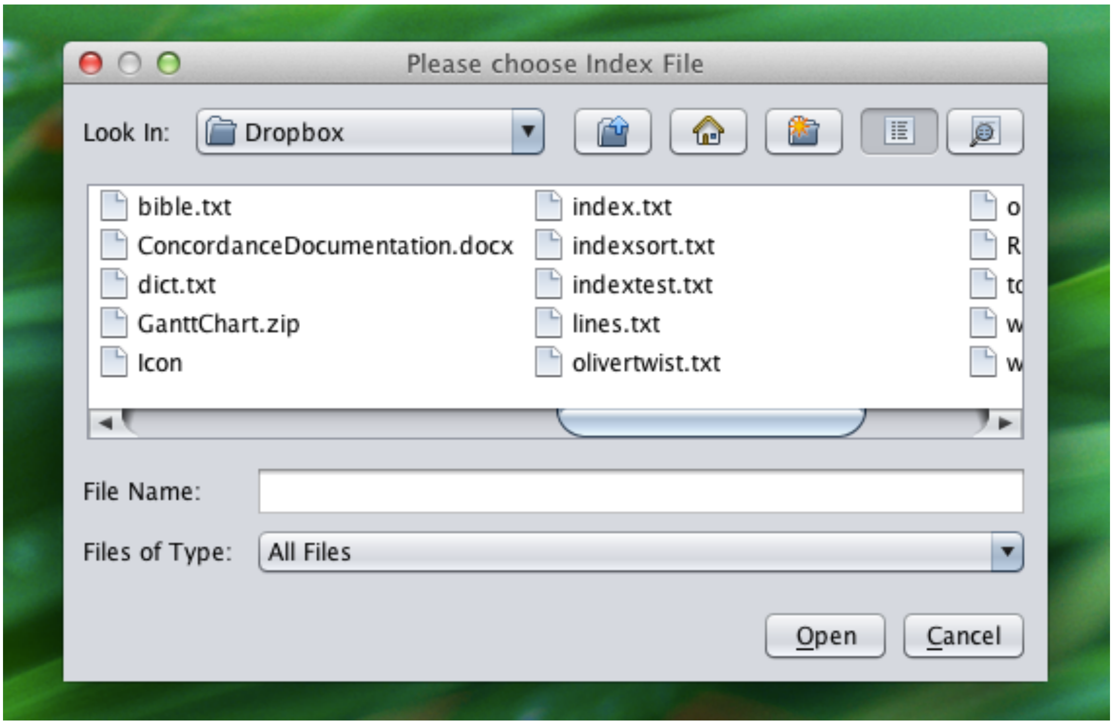
To generate a concordance the user must specify the two text files, an index and a source text.

- Index, a .txt file, each word that is to be indexed should be on its own line.
- Source, a .txt file, any type of text that you want to search.

Once the program has been opened the user should locate their index.txt and source.txt file using the corresponding load button that will display a file chooser.



The file chooser will provide a way to specifying the files you wish to use.
Find the file you want to use then either press return or open.



Look In: Dropbox

File Name:

Files of Type: All Files

A 5000 word index would typically take a few seconds to generate a Concordance of a source text with around 20000 lines, small index files will process almost instantly even with large source texts.

Data Structure Design:

The main Data Structure in use for this program is a Hash Table, this is required in the specification, a Hash Table is the correct Data Structure for the problem because if correctly implemented, the performance of each Hash Table lookup does not increase based on the number of items in the Data Structure. This makes it suitable for a problem like the Concordance, as you are able to index, process and lookup huge amounts of data while maintaining a high level of performance.

In order to fulfill the criteria outlined in the specification another Data Structure is required to store additional information about the words that are being indexed, including their line number and surrounding context.

Some of the options available are as follows:

A Linked List of type Integer that would be used to reference the line number for each index word, this is simple and effective but would not allow for contextual information to be sorted. A Linked List is a good choice as it is easy to work with, self-resizing and is also an efficient method of data storage. I have used the built in Linked List as it would realistically be more feature complete than if I was have to written it myself.

Another option would be to use a Linked List of type String and then use a StringBuffer to build up a string with a line separator. The Integer.toString() function of line number could be called to convert the line number to a String, the line number String can then be added to a StringBuffer followed by a new line character.
The context can then be added on another line and the String.split() method with a regular expression can be used to retrieve and convert the two Strings into a useable format. This is a viable option but does require more overhead as many additional methods will need to be called to present the data in a functioning state, this would not be the best option as it would significantly degrade the performance of the program if a large index file was used.

An alternative to a Linked List would be to use an ArrayList however they are incredibly inefficient and would not be viable for storing and accessing the large amount of data that we may need to store.

It would also be possible to use an Array as the second Data Structure. An Array that is fully populated would be more efficient and thus perform better than a Linked List as Arrays do not require a Link Node to the next value. However using an Array that is a static structure would not fit well with the very dynamic approach of this problem.

The option that I felt was the best was to create a new Class called ConcordanceResult, this class consists of two primitive instance variables, String context and int lineNumber, this class also contains getter and setter methods to allow us to access the values stored in the object. A Linked List was chosen as the Data Structure to store the context and line number, in this case the Linked List being of type ConcordanceResult. Using a custom object to store the line number and context, was in my opinion the most effective option as you can retrieve both line number and context using a simple getter and setter. This maintains performance and a higher level of cohesion.

The way that the main indexing algorithm has been designed is largely based on the approach that I wanted to use for storing the data. Using a Linked List for storage has allowed me to implement a reasonably simple algorithm that performs well yet is easy to read, maintain and improve.

UML and Class Design:

The Java Concordance program is split into 3 packages,

- Concordance – Contains the main data classes and algorithms
- GUI – Provides the classes that implement the Graphical User Interface.
- Tests – Contains JUnit tests to ensure that the program functions as required

I chose to implement the following classes to solve the Concordance problem,

- Concordance Package:
- Concordance – Provides the main method that starts the Graphical User Interface
- FileInputReader – Contains 2 processing algorithms for index words and context.
- Conversion – Removes punctuation to aid indexing.
- ConcordanceResult – Creates object containing context and line number.

- GUI Package:
- ConcordanceFrame – JFrame with various swing components for user interaction.
- CustomModelLineNumbers – Custom JList model for line numbers.
- CustomModel – Custom JList model for index words.

Class Discussion:

The Concordance class is designed only to provide the main method, the main method in this case calls the startGUI() method in ConcordanceFrame and also display a JOptionPane showing the user how to use the program.

FileInputReader is the class that provides the main functionality for the Concordance program, this class contains the instance variables int lineNumber, int LINE_DEFAULT and HashTable indexTable.
Line number just holds the line number that is incremented using a loop, LINE_DEFAULT ensures that line number is always set to 1, this will make sure that the correct line numbers are stored relating to the reference text and is set when the constructor is called. The indexTable is of type HashTable, the primary attribute is of type String and the secondary Data Structure is a LinkedList of type ConcordanceResult.
The following methods are contained in this class:

- processInputIndex(String indexText)
- processInputText(String text)
- returnHashTable();
- returnList();
- getResultsList();

ProcessInputIndex() acts as the first part of a two stage algorithm, its main purpose is to process an input file containing index words and store them in the HashTable. This method then calls the ProcessInputText() method.

ProcessInputText() is the second part of the algorithm, at this stage the HashTable contains the desired index words with a default line number, this method then checks if an index word is contained in the text, if it is then it stores each line number that the index word can be found along with its context, inside the HashTable in a LinkedList.

ReturnHashTable() returns the indexTable HashTable complete with index, line numbers and context.

ReturnList() Uses collections.sort() to determine the correct alphabetical order in which the index words are to be displayed in a JList.
Class Discussion Continued:

GetResultsList() returns the specific LinkedList containing the line numbers and context for a specific index word determined by its specific index value in a JList.

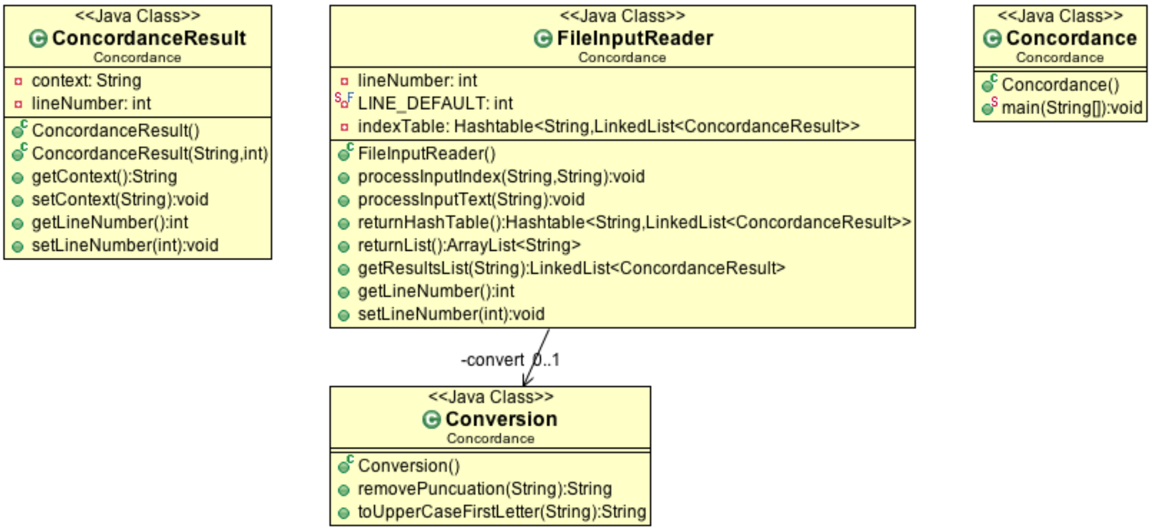
The Conversion class provides a method to remove punctuation from a String, this removes issues with punctuation as the index words and each word in the reference text are converted to lower case and have all non letter characters removed. This method could have been included in any class but for versatility and reuse I felt it was a better to give it its own class, this allows the method to be called globally.
A method to convert the first character of an index word to upper case was also included in this class but is never called as the performance hit when using large files was too severe.

ConcordanceResult class is effectively a wrapper for the two instance variables that contain the line number and context. This is a very effective way of being able to store and access values, using only getter and setter methods, this helps with performance as no complex methods are being called to present the data in a usable state.

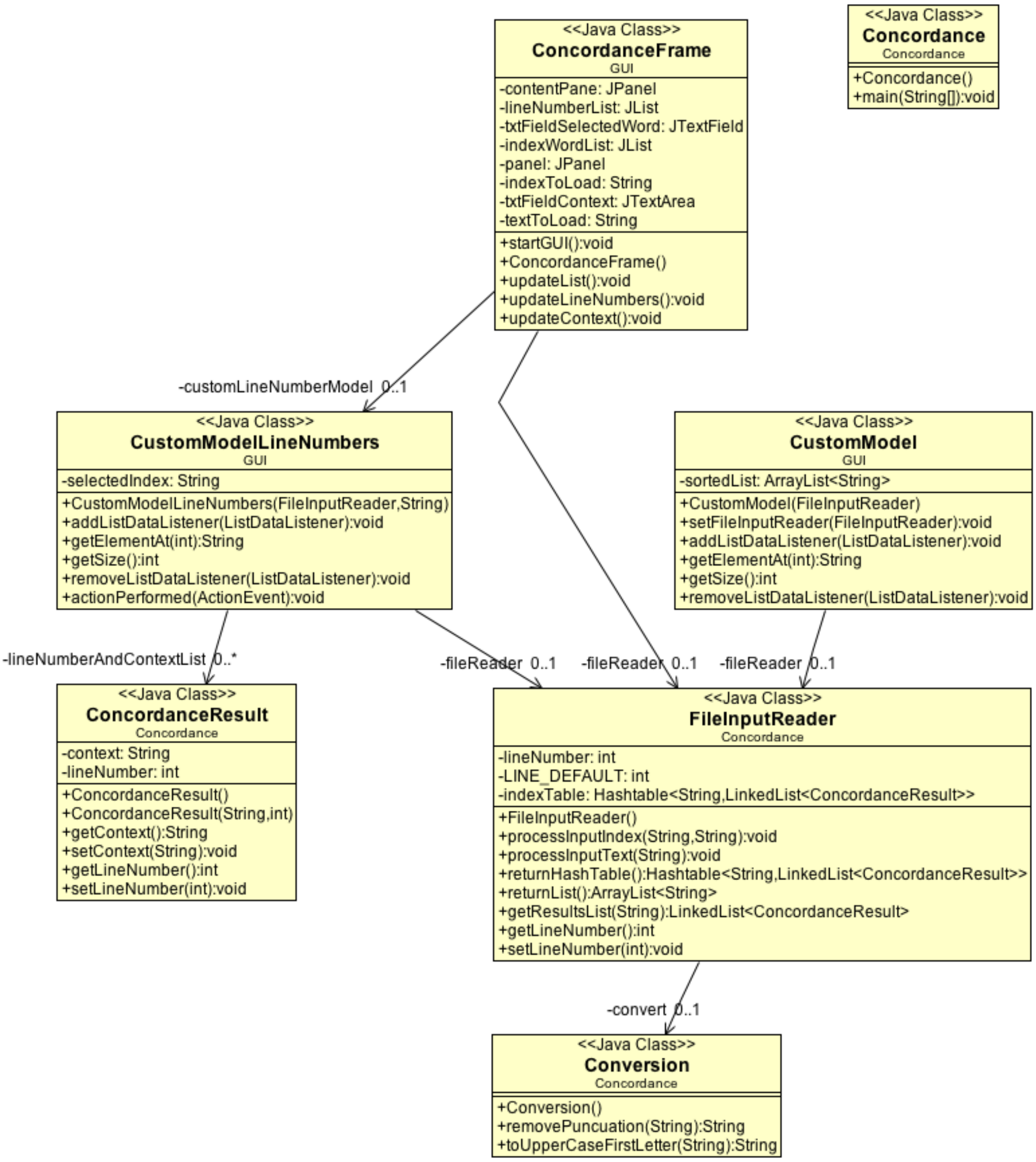
ConcordanceFrame is a JFrame that contains the swing components that are required to display and allow interaction of the GUI with the user. In this case I felt the best approach was to use 2 JLists to display the index words and then the line numbers. Using a ValueChangeListener the users selected word is displayed in a JTextField, context is shown in JTextArea. Two JButtons are also provided to allow the user to pick and choose index and reference texts as they wish.

The CustomModelLineNumbers and CustomModel class are required to create a dynamically generated JList, we do not know the values for each index word or reference text as to provide a static array. To do this two custom table models have been created to ensure that the JList is created dynamically and displays the index and line number information correctly to the user.

I feel that the above design is a viable solution to the Concordance problem, I also feel that is easy to maintain and improve if need be. The GUI also provides a functional and helpful way of being able to quickly find the correct line number and context for the selected index word.
UML Class Diagram of classes specific to generating Concordance:

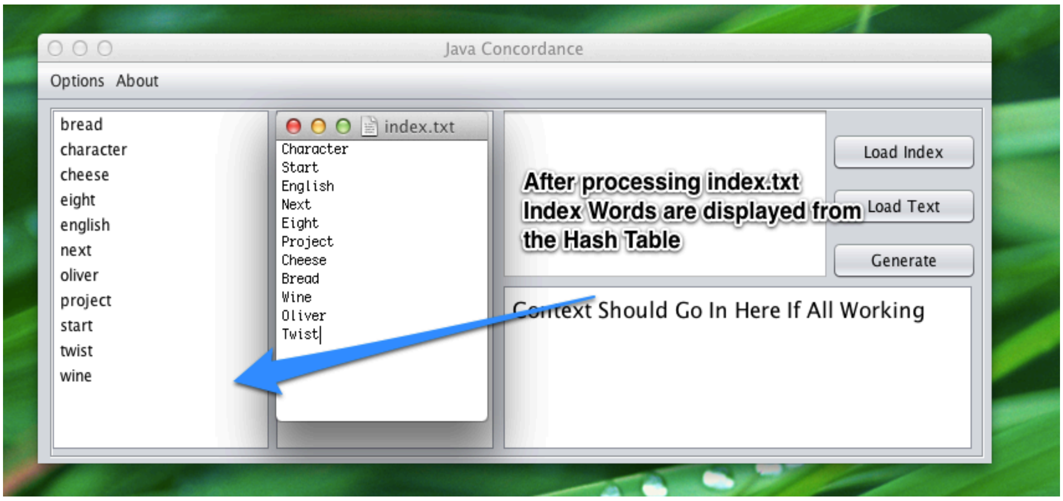


UML Diagram of all classes:


<<Java Class>>
ConcordanceFrame
GUI
-contentPane: JPanel
-lineNumberList: JList
-txtFieldSelectedWord: JTextField
-indexWordList: JList
-panel: JPanel
-indexToLoad: String
-txtFieldContext: JTextArea
-textToLoad: String
+startGUI():void
+ConcordanceFrame()
+updateList():void
+updateLineNumbers():void
+updateContext():void
<<Java Class>>
Concordance
Concordance
+Concordance()
+main(String[]):void
-customLineNumberModel 0..1
<<Java Class>>
CustomModelLineNumbers
GUI
-selectedIndex: String
+CustomModelLineNumbers(FileInputReader,String)
+addListDataListener(ListDataListener):void
+getElementAt(int):String
+getSize():int
+removeListDataListener(ListDataListener):void
+actionPerformed(ActionEvent):void
<<Java Class>>
CustomModel
GUI
-sortedList: ArrayList<String>
+CustomModel(FileInputReader)
+setFileInputReader(FileInputReader):void
+addListDataListener(ListDataListener):void
+getElementAt(int):String
+getSize():int
+removeListDataListener(ListDataListener):void
-lineNumberAndContextList 0..*
-fileReader 0..1
-fileReader 0..1
-fileReader 0..1
<<Java Class>>
ConcordanceResult
Concordance
-context: String
-lineNumber: int
+ConcordanceResult()
+ConcordanceResult(String,int)
+getContext():String
+setContext(String):void
+getLineNumber():int
+setLineNumber(int):void
<<Java Class>>
FileInputReader
Concordance
-lineNumber: int
-LINE_DEFAULT: int
-indexTable: Hashtable<String,LinkedList<ConcordanceResult>>
+FileInputReader()
+processInputIndex(String,String):void
+processInputText(String):void
+returnHashTable():Hashtable<String,LinkedList<ConcordanceResult>>
+returnList():ArrayList<String>
+getResultsList(String):LinkedList<ConcordanceResult>
+getLineNumber():int
+setLineNumber(int):void
-convert 0..1
<<Java Class>>
Conversion
Concordance
+Conversion()
+removePuncuation(String):String
+toUpperCaseFirstLetter(String):String

Functional Requirement	Description	Inputs	Expected Outputs	Pass /Fail	Comments
Read an Index File into a Data Structure	Process and Input Index text file and read the file into a data structure	Text File - Index	Populated Hashtable containing Index Words – Shown via GUI as JList	Pass	Also found in JUnit testInputIndex() Screenshot 0.1
Search a source text for words from the index file	Check a source text for index words contained in a data stucture	Text File – Index Text File – Source HashTable - Search	Line numbers to be displayed along with context	Pass	Also found in JUnit InputTextConcordance() Screenshot 0.2
Present an alphabetical list of index words, with the line numbers of where they are found in the source text	Retrieve unsorted index words from Hash Table and display they sorted alphabetically with line numbers.	Text File – Index Text File – Source HashTable-Search Sorting Algorithm	Index words to be displayed in alphabetical order. Line numbers to be shown for each index word. Context to also be shown.	Pass	Screenshot 0.2

Screenshot 0.1



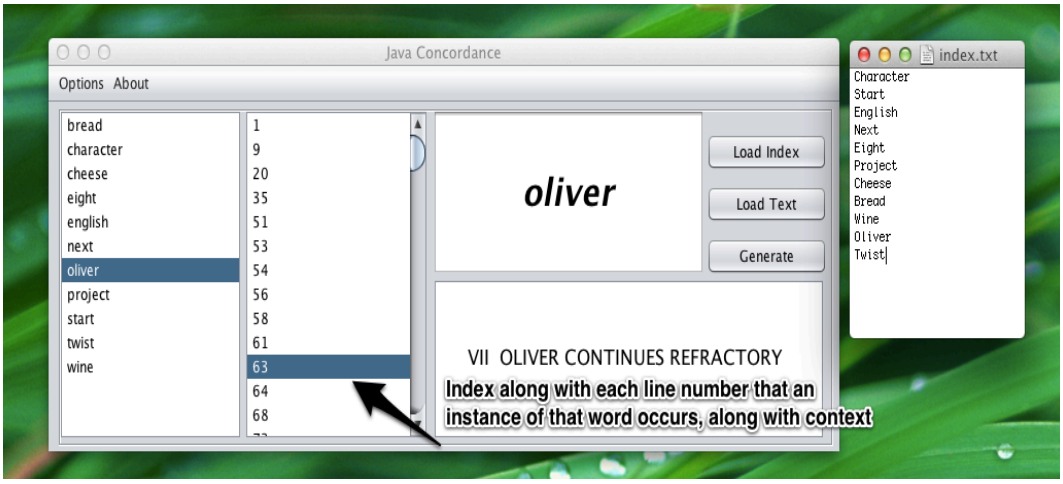
bread
character
cheese
eight
english
next
oliver
project
start
twist
wine

Character
Start
English
Next
Eight
Project
Cheese
Bread
Wine
Oliver
Twist

After processing index.txt Index Words are displayed from the Hash Table

Context Should Go In Here If All Working

Screenshot 0.2



bread
character
cheese
eight
english
next
oliver
project
start
twist
wine

1
9
20
35
51
53
54
56
58
61
63
64
68

oliver

VII OLIVER CONTINUES REFRACTORY

Index along with each line number that an instance of that word occurs, along with context

Character
Start
English
Next
Eight
Project
Cheese
Bread
Wine
Oliver
Twist

Algorithm Design and Pseudo code:

Algoirthm Name – processInputIndex(String indexToLoad, String textToLoad)
Algorithm Purpose – Read an index file into a data structure

Take two parameters, one is a list of index words and the other a source text file, both in .txt;

Read in .txt file containing index words;

Create temporary String to hold a line of the text file, a single index word;

While there is a line left to read in the text file, create an empty Linked List and add the index word String and the empty Linked List to the Hash Table;

Close the File Reader;

If an IOException was thrown, print a stack trace;

Call the method to compare the index words to the source text using second parameter.

Algorithm Name – processInputText(String,textToLoad)
Algorithm Purpose – Search a source text for words from the index file

Take one parameter, a String path to a .txt file containing source text;

Prepare to read in .txt file containing source text;

Create temporary String to hold a line of the text file;

While there is a line left to read in the text file, split the sentence into individual words;

While there are words remaining, create String to hold a word, remove punctuation and convert the word to lower case;

If any word read in from the source text is contained inside the Index Hash Table, create a Linked List of ConcordanceResult objects, add all the words on the line and add the line number to the Linked List, then add to Hash Table;

Increment line number by one;

Close the File Reader;

Algorithm Name – returnList()
Algorithm Purpose – Return a index files from Hash Table in alphabetical order

Create an Array List;

For every index word in the Hash Table, add it to the Array List;

Sort ArrayList by alphabetical order;

Return Array List;

** Array List only used as it is easy to sort, it would be a bad idea to use one as a primary data structure, due to its poor performance

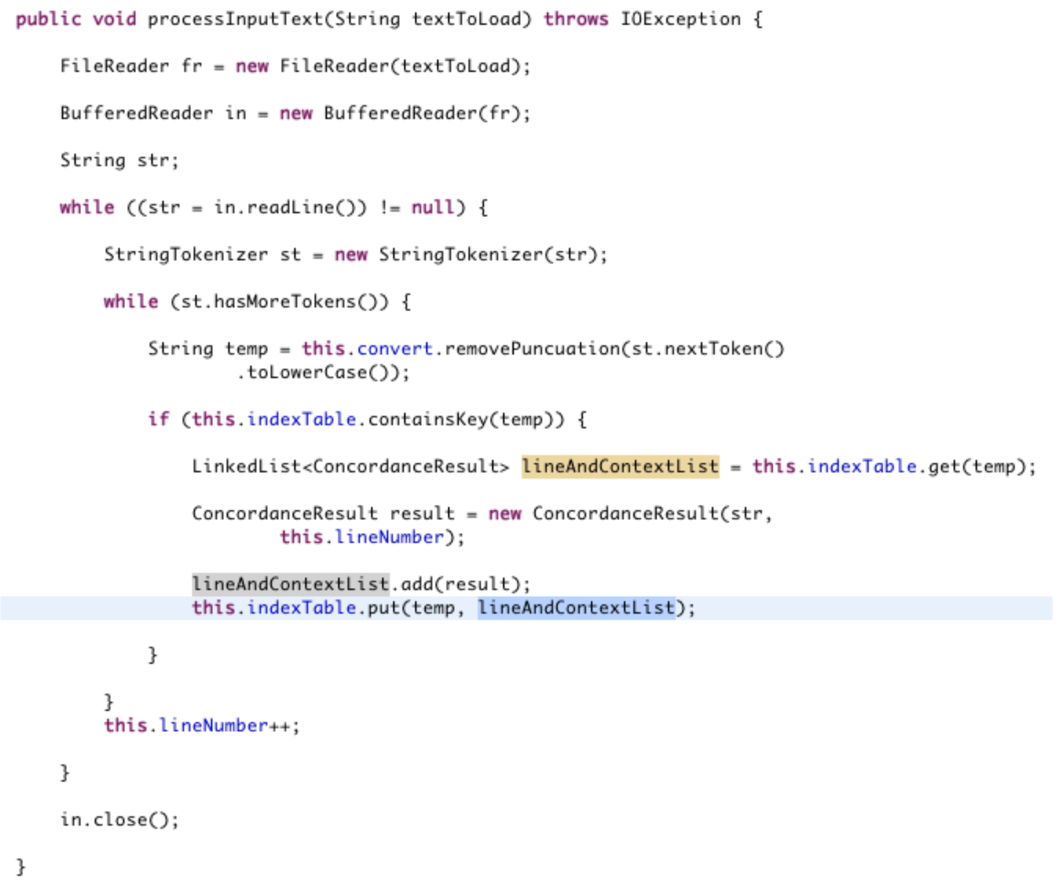
Algorithm Discussion:

The purpose of this algorithm is to process a source text file and find occurrences of an index word, inside the source text and record the line number and context surround the index word.

To do this A FileReader object is created using a String containing the filename of the source text to be read in. A BufferedReader is then created to allow the text file to be read in line by line, this also serves an operator for a while loop that will continue reading until there is no lines remaining for the BufferedReader to process.

A StringTokenizer object is then created to tokenize(split sentence into words) the sentence being read in. Another while loop ensures that while there tokens left to be processed they are converted to lower case and have their punctuation removed. This allows for more accurate Concordance to be generated as it makes no difference if an index word is capitalised or contains punctuation or not.

An If statement checks if any token is contained in the Hash Table of Index words. If a token matches an Index word then each line number and its context are added to a Linked List data structure and then added back into the Hash Table under the key belonging to the Index word. The line number is finally incremented as that is the end of one complete pass of the first while loop, once every line and token has been processed the File Reader is then closed to ensure good memory management.



```
public void processInputText(String textToLoad) throws IOException {

    FileReader fr = new FileReader(textToLoad);

    BufferedReader in = new BufferedReader(fr);

    String str;

    while ((str = in.readLine()) != null) {

        StringTokenizer st = new StringTokenizer(str);

        while (st.hasMoreTokens()) {

            String temp = this.convert.removePuncuation(st.nextToken()
                    .toLowerCase());

            if (this.indexTable.containsKey(temp)) {

                LinkedList<ConcordanceResult> lineAndContextList = this.indexTable.get(temp);

                ConcordanceResult result = new ConcordanceResult(str,
                        this.lineNumber);

                lineAndContextList.add(result);
                this.indexTable.put(temp, lineAndContextList);

            }

        }
        this.lineNumber++;

    }

    in.close();

}
```