Lab 16 Dictionary Client

Goal

In this lab you will use a dictionary in an animated application to highlight words that could be misspelled in a text file.

Resources

- Chapter 19: Dictionaries
- Appendix E: File Input and Output (online)
- *docs.oracle.com/javase/8/docs/api/*—API documentation for the `Scanner` class
- *docs.oracle.com/javase/8/docs/api/*—API documentation for the `StringTokenizer` class
- *Spell.jar*—The final animated application
- Lab Manual Appendix —An Animation Framework

In `javadoc` directory

- *DictionaryInterface.html*—API documentation for the `DictionaryInterface` ADT
- *Wordlet.html*—API documentation for a class representing a word with a flag for whether it is spelled correctly.
- *LinesToDisplay.html*—API documentation for a class representing a number of lines to be displayed graphically.

Java Files

- *FindDefaultDirectory.java*
- *LinesToDisplay.java*
- *MisspellActionThread.java*
- *MisspellApplication.java*
- *Wordlet.java*

There are other files used to animate the application. For a full description, see Appendix: An Animation Framework of this manual.

In *DictionaryPackage* directory

- `DictionaryInterface.java`
- `HashedMapAdaptor.java`

Input Files

- *check.txt*—A short text file to check for possible spelling errors
- *sampleDictionary.txt*—A small dictionary

Introduction

The dictionary ADT is set of associations between two items: the key and the value. A concrete example is a dictionary, such as the *Oxford English Dictionary*. It associates words with their definitions. Given a key (word), you can find its value (definition).

There are a number of ways you could implement the dictionary ADT. In this lab a hash table will be used to implement the dictionary. The details of how hash tables work will be considered in depth later (Chapters 21 and 22). For now, the important features of a hash table are that they allow fast access and that the items in the hash table are not ordered by their keys.

Pre-Lab Visualization

Loading the Dictionary

Given a dictionary of words and a word to be checked for spelling, what is the key? (What is being searched for?). What is the value? (What do we need to know about the key?)

Reading the file that contains correctly spelled words into the dictionary requires that the file must be parsed (the file must be broken up into pieces each containing a single word). Sometimes the format of the file will be tightly specified. In this case, the format will be pretty loose. The correct words will be in a file. They will be separated by either space or return. Review Appendix E of the book and the API for `Scanner`.

Write an algorithm for reading the words from a file. Assume that the file is already opened as an instance of `Scanner` with the name `input`.

Wordlet Class

As the spelling checker executes, it will need to consider each word in the text. As it decides whether a word is spelled correctly or not, that information will need to be associated with the word. The function of the `Wordlet` class is to remember whether the word is spelled correctly.

The `Wordlet` class holds a chunk of text (presumed to be a word) and a `boolean` variable indicating if the word is spelled correctly. Review that class if you have not done so already.

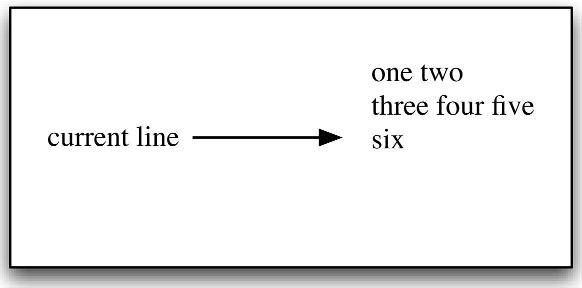
LinesToDisplay Class

A data structure is needed to hold the lines of wordlets that will be displayed by the animated application. Here are the requirements for the class.

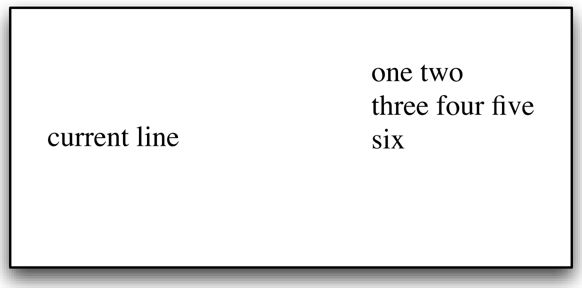
1. It must remember up to 10 lines of checked text.
2. It must know whether words are spelled correctly.
3. It must have a line of text that it is currently composing.
4. It must be able to add a wordlet to the current line.
5. It must be able to move to the next line of text.

Consider the wordlets in a single line. Would an array or a list be preferable for storing them?

Suppose that the initial state of a lines to display object is

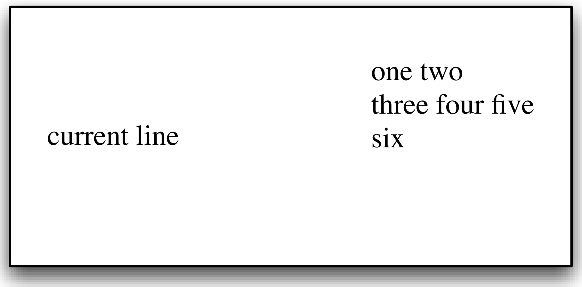


Show the new state if the wordlet " " (a single blank) is added.



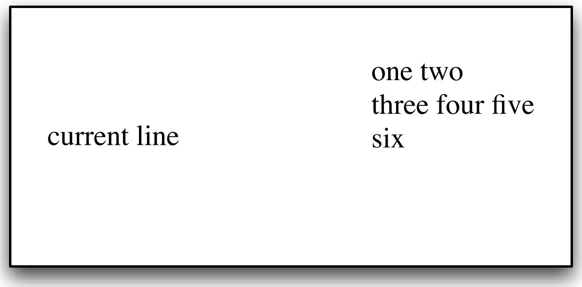
What should be displayed in the animation?

Show the new state if the wordlet “seven” is added instead.



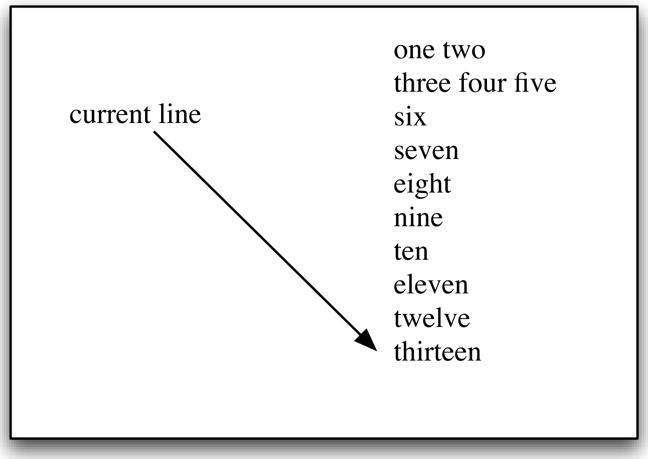
What should be displayed in the animation?

Show the new state if the current line is ended.

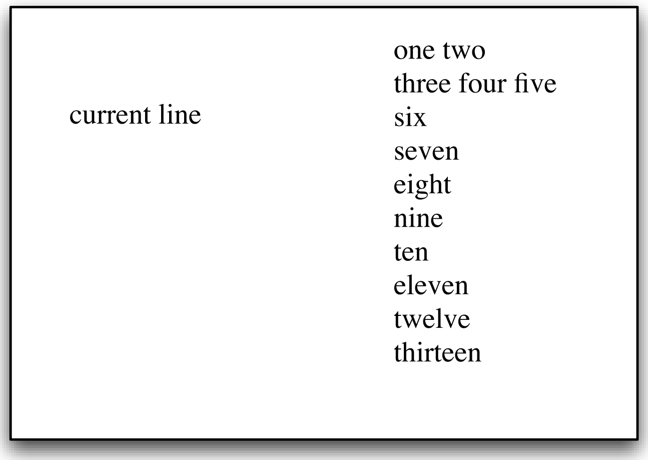


What should be displayed in the animation?

Suppose that the initial state is:



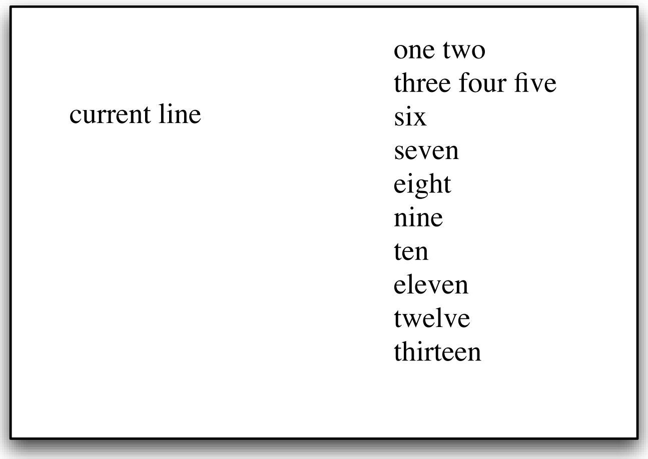
Show the new state if the line is ended and then the wordlet “fourteen” is added.



Are there more than ten lines of text to be displayed?

What should be displayed in the animation?

Show the new state if the line is ended and then the wordlet “fifteen” is added.



Are there more than ten lines of text to be displayed?

What should be displayed in the animation?

Given that the private state variables of the class are:
```
      public static final int LINES = 10; // Display 10 lines
      private ArrayList<Wordlet> lines[];
      private int currentLine;
```

Give an algorithm for adding a wordlet to the current line.

Give an algorithm for moving to the next line.

Reading Words

While parsing the words in the text file is similar to reading the words from the dictionary, it will turn out that using a `StringTokenizer` will make the job easier. A double loop will be employed. A `Scanner` will be used to read the lines and a `StringTokenizer` will be used to break up each line. Consider the following lines of text.

```
      Just some fun text! (You shouldn't take it seriously.)

      Slowly she turned and said, "Bob, I don't know

      if I like you any more.  Do you know why?"

      He replied "As the raven once said: 'Never more'".
```

Draw boxes around all the words.

What characters in the text indicated the beginning or end of a word (delimiters)?

Are there other characters that might be delimiters?

In our application, we would like to mark words that could be incorrectly spelled within a given body of text. This means that we will not be able to just read words (and discard the delimeters), but will need to make sure that every character is in some wordlet.

Review the API for `StringTokenizer` and show how one can be created that satisfies the preceding observations.

Give an algorithm that will read a file and create wordlets. Add each wordlet to the lines display. At the end of every line, go to the next line in the lines display. Do not worry about checking the spelling.

Algorithm:

Checking the Spelling

Some of the words that the string tokenizer will produce will be things like "!" and "120". The application will assume that if a word has no alpha characters in it, it should not be marked as incorrect. (An alpha character is a lower or upper case letter.)

Write an algorithm that given a string will return true if all the characters are non alpha characters. The `isLetter()` method in the `Character` class can be useful here.

Directed Lab Work

Loading the Dictionary

All of the classes needed for the `MisspellApplication` exist. Some of them need to be completed. This application is based on the animated application framework. If you have not already, you should look at the description of it in Appendix: An Animation Framework of this manual. The classes that you will be completing are `LinesToDisplay` and `MisspellActionThread`. The `Wordlet` class is also specific to this application, but completely defined. Take a look at that code now if you have not done so already.

Today's application will read from two files. Different Java run time environments use different directories as their default when opening a file. It might be the directory that the class is in or it may be somewhere else. First, we would like to find where the default directory is. (Provided that you have not already done so for the Maze application from Lab 10.)

Step 1. Compile and then run `main` from the class `FindDefaultDirectory`.

Step 2. Leave your Java environment temporarily and search for the file name *`DefDirHere.txt`*.

Step 3. Move the files *`sampleDictionary.txt`* and *`check.txt`* to the directory your particular implementation of Java reads from and writes to.

Step 4. Compile the class `MisspellApplication`. Run the `main` method in `MisspellApplication`.

Checkpoint: If all has gone well, you should get a graphical user interface with step controls along the top and application setup controls on the bottom. There should be two text fields where you can enter the name of a file containing the dictionary words and another file that contains the text to be checked. Type check.txt for the text file and then press enter. There should be a message indicating that it is now the text file. If not, check to make sure that you copied the file to the correct place. Type sampleDictionary.txt for the dictionary file and then press enter. Again there should be a message confirming the file is readable.

Step 5. In the method `loadDictionary()` in `MisspellActionThread`, add code that will read the words and put them into the dictionary. The dictionary file contains words that are either separated by spaces or lines. A single loop is needed. Refer back to the algorithm you wrote for the pre-lab exercises and complete the code.

Step 6. Just after reading in all the words, print the dictionary to `System.out`.

Step 7. In the method `executeApplication`, add a call to `loadDictionary()`. Immediately after, set the variable `dictionaryLoaded` to `true`. When the application draws itself, it will now indicate that the dictionary has been loaded. Add in the following line of code to make it pause before continuing. (For questions about the function of this method, see the discussion in Appendix: An Animation Framework.)

```
animationPause();
```

Checkpoint: Press step twice. The display should indicate that the dictionary was loaded. The dictionary should be printed on output. Check it against the file.

Completing LinesToDisplay

Step 8. Complete the constructor for the class `LinesToDisplay`.

Step 9. Refer to the pre-lab exercises and complete the code for the method `addWordlet()`.

Step 10. Refer to the pre-lab exercises and complete the code for the method `nextLine()`.

Step 11. In `executeApplication` in `MisspellActionThread`, add the following four lines of code.

```
    myLines.addWordlet(new Wordlet("abc", true));
    myLines.nextLine();
    myLines.addWordlet(new Wordlet("def", false));
    myLines.nextLine();
```

Checkpoint: Press step three times. The wordlet abc should be in black on one line and def should be in red on the next line. If not, debug the code and retest.

Step 12. Comment out the four lines of code entered in the previous step.

The next goal is to parse the text file into wordlets and put them in the display. For now, even though we call a method to check the spelling, it will always return false. Thus, all words will be considered to have an incorrect spelling.

Reading Words

Step 13. Refer to the pre-lab exercises and complete the code in the method `checkWords()`. As each wordlet is created, use `checkWord()` to determine if the spelling is correct. (It will always return false until we change that in the next part.)

Step 14. For the animation, add in the line

```
    animationPause();
```

just after the call to `nextLine()`.
(We could put in a pause after adding every wordlet. What effect would this have on the application?)

Checkpoint: Step the application. Each line in the text file should appear exactly with all the text in red. If not, debug the code and retest.

The final goal is to check the spelling of the wordlets.

Checking the Spelling

Step 15. Complete the `checkWord()` method to search for the word in the dictionary. Return true if the word is in the dictionary.

Checkpoint: Step the application. Each line in the text file should appear exactly. The words in the dictionary should now appear in black.

Step 16. It would be nice if the punctuation did not show up in red. Refer to the algorithm from the pre-lab exercises and add code to `checkWord()` to return true if the word consists only of punctuation characters.

Final checkpoint: Step the application. Most of the punctuation should now be in black.

There are a number of improvements that can be made to this application. See the post-lab exercises for some of them.

Post-Lab Follow-Ups

1. The program does not correctly check proper words that start with an uppercase letter. Modify the spelling checker so that any word in the dictionary that starts in uppercase must always start in uppercase. Any other words may start either in upper or lowercase. This should not affect the display of the lines of text.

2. The program does not correctly handle words that have an apostrophe. Use a tokenizer that does not split on apostrophes. Instead check the beginning and end of the words for nonalpha characters and split those off into wordlets of their own. This should not affect the display of the lines of text.

3. The program does not handle words that have been hyphenated and broken across lines. Check for this situation and combine the two pieces of the hyphenated word and place it on the next line.

4. Write a program that will do automatic correction of text. Read a file that contains sets of words each on a single line. The first word is the correct word and the rest of the words on that line are common misspellings. Display the corrected text with any changed words in green.

5. Write a program that will highlight key words in a java program in blue. Make sure that key words inside of comments are not highlighted.

6. Use a dictionary to add memoization to the `better()` method in `RecursiveFibonacci`. In memoization, the method checks to see if it has the answer stored in the dictionary. If it does, it will just return the answer. If not, it will do the computation and then, before returning the answer, remember it in the dictionary.