Assignment 4: Readability Indices

Thanks to Sophia Westwood for suggesting Flesch-Kincaid readability as an assignment, to Eric Roberts for the syllable-counting heuristic, and to the CS106A section leaders for preflighting the assignment. Thanks to Leonid Shamis for suggesting the libraries used in the web scraper. This assignment uses the Readability API (www.readability.com), the Jsoup library, and the simple-json library.

Many word processing programs (such as Microsoft Word) employ *readability indices* that estimate how difficult it is to read passages of text. Groups that produce written materials for wide audiences (non-profit groups, government agencies, etc.) often use these readability indices to check that the materials they produce aren't too complex for a general audience.

In this assignment, you'll implement two readability indices. In the process, you'll get a much stronger understanding of strings, decomposition, testing, debugging, file processing, and `ArrayLists`.

Due Wednesday, February 19 at 3:15PM

Assignment Review Hours: Thursday, February 13th, 5:30PM – 6:30PM in Hewlett 200

Assignment Overview

As of 2014, computers are notoriously bad at reading text. Even the most advanced algorithms for processing text still get tripped up on simple examples. Given this, readability indices work using *heuristics*, approaches for solving a problem that, while often accurate, are not always correct. In this assignment, you'll implement two standard heuristics for estimating text complexity: the *Flesch-Kincaid grade level test* and the *Dale-Chall readability score*.

The *Flesch-Kincaid grade level test* estimates at what grade level a reader would have to be in order to comprehend a text. For example, a passage with Flesch-Kincaid grade level of 5.15 could be read by a typical fifth-grader, while a passage with grade level 15.1 would be appropriate for a typical college junior. The Flesch-Kincaid grade level score is evaluated by counting the number of words, sentences, and syllables in a piece of text, then computing

$$Grade = C_0 + C_1\left(\frac{num\ words}{num\ sentences}\right) + C_2\left(\frac{num\ syllables}{num\ words}\right)$$

where C_0, C_1, and C_2 are constants given later in this handout. Intuitively, more complicated texts tend to have longer sentences (therefore, more average words per sentence) and longer words (therefore, more average syllables per word), so the above value is often a good approximation of the actual readaibility. Notice that Flesch-Kincaid makes no attempt whatsoever to determine what the text actually says; it just examines simple structural properties of that text and extrapolates from that.

The *Dale-Chall Readability Formula* works along the same lines. It's evaluated by counting up the number of total words, sentences, and “difficult words” in the text, then computing

$$Difficulty = D_0\left(\frac{num\ difficult\ words}{num\ words} \times D_1\right) + D_2\left(\frac{num\ words}{num\ sentences}\right) + D_3\,bonus$$

Here, D_0, D_1, D_2, D_3, and *bonus* are described later in this handout. As before, since complicated texts tend to have long sentences and difficult words, this formula works well in practice.

Whereas the Flesch-Kincaid score directly maps to an estimated grade level, the Dale-Chall readability index evaluates to a numerical score that can be interpreted according to the following table:[*]

Score	Difficulty
0 – 5	Readable by an average 4th grader.
5 – 6	Readable by an average 5th or 6th grader.
6 – 7	Readable by an average 7th or 8th grader.
7 – 8	Readable by an average 9th or 10th grader.
8 – 9	Readable by an average 11th or 12th grader.
9 – 10	Readable by an average college student.
10+	Readable by an average college graduate.

As with Flesch-Kincaid, notice that the Dale-Chall readability test doesn't make any attempt to actually understand the text it reads. It just computes simple statistics and bases its guess on that.

We've broken this assignment down into seven smaller steps, each of which you can design and test in isolation. Once you've completed all of them, you'll have a slick, clean piece of software that can estimate reading complexities.

[*] Source: http://en.wikipedia.org/wiki/Dale-Chall_Readability_Formula.

Step One: Estimate Syllables in a Word

In order to compute the Flesch-Kincaid or Dale-Chall scores for a piece of text, you'll need to be able to count up the number of syllables in a word. Your job in this step is to write a method

```
private int syllablesInWord(String word)
```

that takes in a single word and returns the number of syllables in that word.

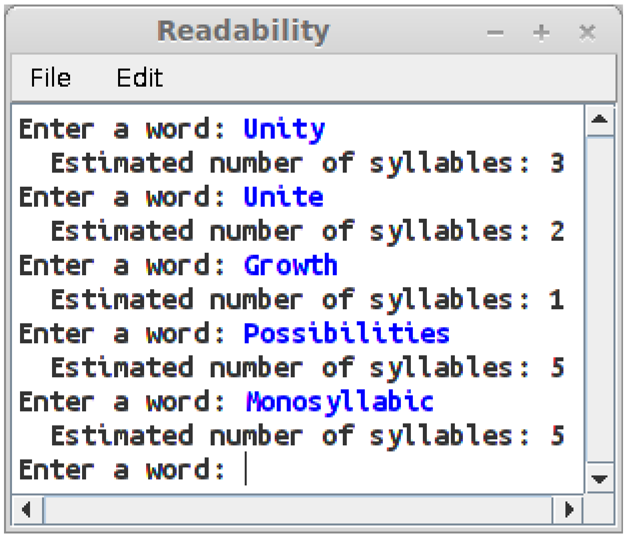
It is difficult to determine the exact number of syllables in a word just from its spelling. For example, the word `are` has just one syllable, while the similarly-spelled `area` has three. Therefore, we aren't going to ask you to count syllables exactly. Instead, you should approximate the number of syllables using the following heuristic: count up the number of vowels in the word (including 'y'), *except* for

- Vowels that have vowels directly before them, and
- The letter *e*, if it appears by itself at the end of a word.

Notice that under this definition, the word `me` would have zero syllables in it, since the 'e' at the end of the word doesn't contribute to the total. To address this, *your method should always report that there is at least one syllable in any input word*, even if the heuristic would otherwise report zero. This resulting method will correctly estimate that there are two syllables in `quokka` and two syllables in `springbok`, though it does get the number of syllables in `syllable` wrong (guessing two instead of three). With the zero-syllable correction in place, the heuristic correctly counts syllables in the words `the`, `be`, and `she`.

Test your method thoroughly before moving on. Think about how the heuristic works and choose test words that ensure your code correctly implements the heuristic. When testing, don't worry if the answer your program gives isn't exactly the correct number of syllables in the word (you're using a heuristic, after all), but do make sure that your program produces values that agree with what we've described above.

Here is a sample run of the testing program using our own solution code:



```
Enter a word: Unity
  Estimated number of syllables: 3
Enter a word: Unite
  Estimated number of syllables: 2
Enter a word: Growth
  Estimated number of syllables: 1
Enter a word: Possibilities
  Estimated number of syllables: 5
Enter a word: Monosyllabic
  Estimated number of syllables: 5
Enter a word:
```

These tests aren't exhaustive and you should *definitely* test your program on more sample inputs than shown here.

Question 1 of the assignment writeup asks what specific words you chose as test cases and why you chose them, so we recommend answering that writeup question before moving on to the next part of the assignment.

Step Two: Implement Tokenization

To compute the Flesch-Kincaid or Dale-Chall scores of a document, you will also need to count up how many words and sentences there are in the document. To make it easier to do this, you will use a technique called *tokenization* (also called *lexical analysis* or *scanning*) to break the document down into simpler constituent parts.

For example, suppose that you have the following document:

Time flies like an arrow. Fruit flies like a banana.

If this document is represented as a single string, you won't be able to use your `syllablesInWord` method from Step One of the assignment to count up how many total syllables there are in the document; that method assumes that the input is a single word, not a collection of words. To make it easier to identify where the words are, you will write a function to *tokenize* the string by breaking it down into a number of smaller strings (called *tokens*), each of which represents either a word, space, or punctuation symbol. For example, you might split the above document into these pieces:

$\boxed{\text{Time}}\ \boxed{\ }\ \boxed{\text{flies}}\ \boxed{\ }\ \boxed{\text{like}}\ \boxed{\ }\ \boxed{\text{an}}\ \boxed{\ }\ \boxed{\text{arrow}}\ \boxed{.}\ \boxed{\ }\ \boxed{\text{Fruit}}\ \boxed{\ }\ \boxed{\text{flies}}\ \boxed{\ }\ \boxed{\text{like}}\ \boxed{\ }\ \boxed{\text{a}}\ \boxed{\ }\ \boxed{\text{banana}}\ \boxed{.}$

Notice that each piece is either a word (“Time,” “arrow,” etc.), a punctuation symbol (“.”), or a space character (“ ”). Your job in Step Two of this assignment is to implement a method

```
private ArrayList<String> tokenize(String input)
```

that will accept as input a string that should be cut into tokens and which returns an `ArrayList` holding all of those tokens.

Determining the “right” tokenization of an arbitrary string is difficult, so for this part of the assignment you will use the following heuristics:

- Any consecutive sequence of letters becomes a single token containing those letters.
- Any character that isn't a letter becomes a token consisting of just that character.

For example, the text

I think, therefore I am.

would be tokenized as

$\boxed{\textbf{I}}\ \boxed{\ }\ \boxed{\textbf{think}}\ \boxed{\textbf{,}}\ \boxed{\ }\ \boxed{\textbf{therefore}}\ \boxed{\ }\ \boxed{\textbf{I}}\ \boxed{\ }\ \boxed{\textbf{am}}\ \boxed{\textbf{.}}$

Similarly, the text

I am (because I think).

Should be tokenized as

$\boxed{\textbf{I}}\ \boxed{\ }\ \boxed{\textbf{am}}\ \boxed{\ }\ \boxed{\textbf{(}}\ \boxed{\textbf{because}}\ \boxed{\ }\ \boxed{\textbf{I}}\ \boxed{\ }\ \boxed{\textbf{think}}\ \boxed{\textbf{)}}\ \boxed{\textbf{.}}$

Since this is just a heuristic, there's no guarantee that this will correctly tokenize all strings. For example, under our definition of a token, the string **isn't** would be tokenized as the three tokens: **isn**, **'**, and **t**. Similarly, the string **Wait...** would tokenize as the four tokens **Wait**, **.**, **.**, and **.**.

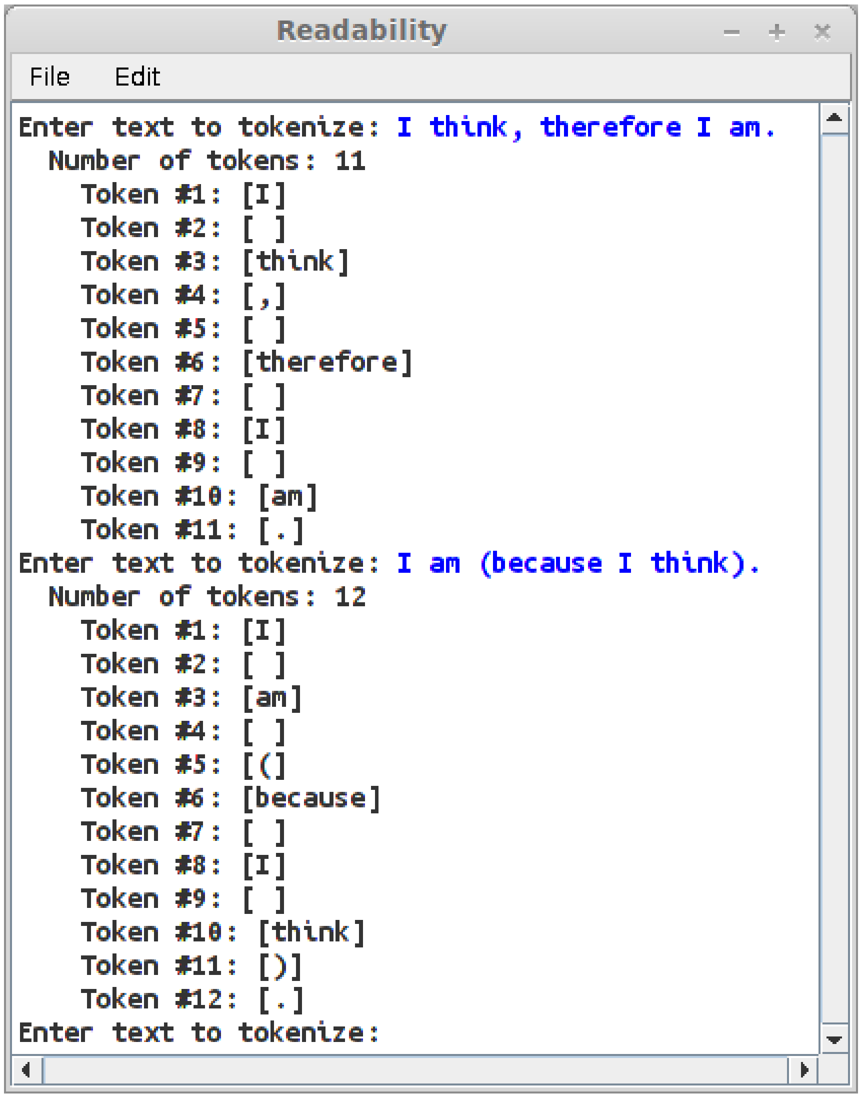
We recommend using the following approach. Read the characters of the string from the left to the right. If the character you read isn't a letter, then that character should be in a token by itself. Otherwise, it's a letter. Keep reading characters from the string until you run out of characters or read a non-letter character. The token consists of all the letter you read up until the end of the string or the non-letter character you found.

Test your method before moving on; there are many cases that your code needs to be able to handle and (based on our own experience coding up this method) you probably will have a few bugs in your initial implementation. We recommend changing your `run` method to the following, which will read in a line of text from the user, tokenize it, then output all of the tokens:

```
public void run() {
    while (true) {
        String line = readLine("Enter text to tokenize: ");
        if (line.isEmpty()) break;

        ArrayList<String> tokens = tokenize(line);
        println("  Number of tokens: " + tokens.size());
        for (int i = 0; i < tokens.size(); i++) {
            println("    Token #" + (i + 1) + ": [" + tokens.get(i) + "]");
        }
    }
}
```

Here's a sample run of the program with correct outputs:



```
Enter text to tokenize: I think, therefore I am.
  Number of tokens: 11
    Token #1: [I]
    Token #2: [ ]
    Token #3: [think]
    Token #4: [,]
    Token #5: [ ]
    Token #6: [therefore]
    Token #7: [ ]
    Token #8: [I]
    Token #9: [ ]
    Token #10: [am]
    Token #11: [.]
Enter text to tokenize: I am (because I think).
  Number of tokens: 12
    Token #1: [I]
    Token #2: [ ]
    Token #3: [am]
    Token #4: [ ]
    Token #5: [(]
    Token #6: [because]
    Token #7: [ ]
    Token #8: [I]
    Token #9: [ ]
    Token #10: [think]
    Token #11: [)]
    Token #12: [.]
Enter text to tokenize:
```

Be sure to test this program on lots of other inputs; there are many cases to consider! **Question 2 of the writeup asks you what specific strings you used to test this method and why you chose them, so we recommend answering that question before moving on to the next part of the assignment.**

Step Three: Count Words, Syllables, and Sentences

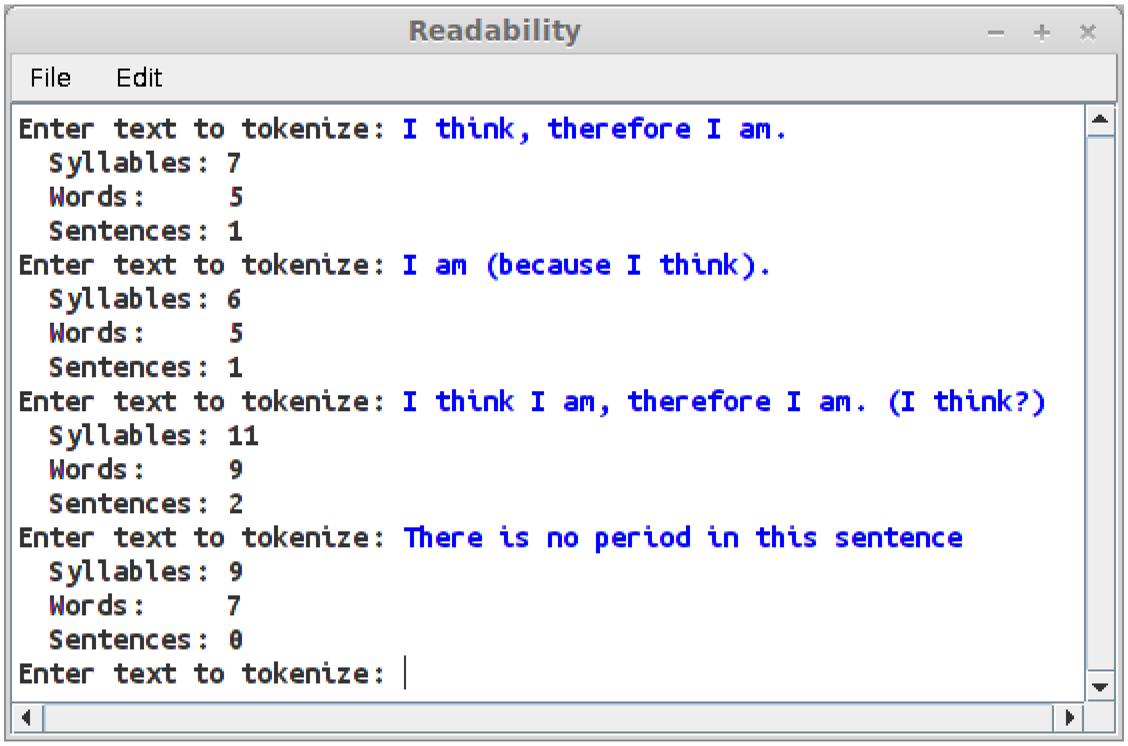
At this point, you've written two major methods: one that counts syllables in a word and one that tokenizes a line. Your job in this Step is to write these three methods:

```
private int syllablesInLine(ArrayList<String> tokens)
private int wordsInLine(ArrayList<String> tokens)
private int sentencesInLine(ArrayList<String> tokens)
```

Each of these methods will accept as input an `ArrayList<String>` containing a tokenized version of a line of text, then will report the total number of syllables in that line, the total number of words in that line, and the total number of sentences in that line, respectively.

To count up the number of syllables in a line of text, you can just sum up the number of syllables in each word in that line. We'll consider a word to be any token that starts with a letter. You can count up the number of words using a similar approach. Finally, you can approximate the number of sentences in that line by adding up the number of times you see a period, question mark, or exclamation point in that line. This isn't a perfect measure, but it should work quite well.

Before moving on, why don't you write some testing code to see if your methods work correctly? We'll leave the task of writing the testing code to you. When we coded up our version of this assignment, we wrote some testing code of our own. Here's a sample output from our testing program with the (correct) counts for each line.



```
Enter text to tokenize: I think, therefore I am.
  Syllables: 7
  Words:     5
  Sentences: 1
Enter text to tokenize: I am (because I think).
  Syllables: 6
  Words:     5
  Sentences: 1
Enter text to tokenize: I think I am, therefore I am. (I think?)
  Syllables: 11
  Words:     9
  Sentences: 2
Enter text to tokenize: There is no period in this sentence
  Syllables: 9
  Words:     7
  Sentences: 0
Enter text to tokenize:
```

Question 3 of the writeup asks you what specific strings you used to test these methods, so we recommend answering that question before moving on to the next part of the assignment.

Step Four: Support File Reading

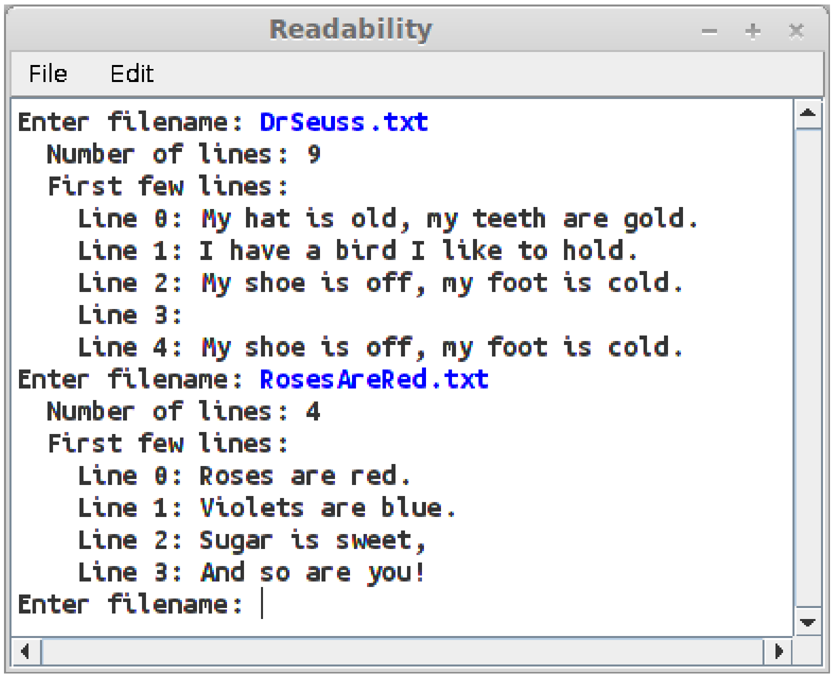
At this point, you're now able to read individual lines of text from the user and compute the number of syllables, words, and sentences from those lines of text. In order to compute the readability index of an entire piece of text, you can just read that text one line at a time, counting up how many syllables, words, and sentences there are in each line, and then use the formulas from the start of the handout to compute the final grade level score.

To do this, you'll write a method that reads the contents of a file. Write a method

```
private ArrayList<String> fileContents(String filename)
```

This method should accept a filename as input, then return an `ArrayList<String>` containing all of the lines of the file. If you can't open the specified file (either because it doesn't exist or because the user typed in the name of the file incorrectly), your method should return `null` to indicate that the file couldn't be read.

As before, be sure to test this program before you move on to the next step. Here's the output of a program that we wrote to test out our own `fileContents` method:



```
Enter filename: DrSeuss.txt
  Number of lines: 9
  First few lines:
    Line 0: My hat is old, my teeth are gold.
    Line 1: I have a bird I like to hold.
    Line 2: My shoe is off, my foot is cold.
    Line 3:
    Line 4: My shoe is off, my foot is cold.
Enter filename: RosesAreRed.txt
  Number of lines: 4
  First few lines:
    Line 0: Roses are red.
    Line 1: Violets are blue.
    Line 2: Sugar is sweet,
    Line 3: And so are you!
Enter filename: 
```

Step Five: Implement the Flesch-Kincaid Grade Level Test

Now that you have all the pieces, it's time to put them together to compute the Flesch-Kincaid grade level score for a piece of text. Your job is to write the following method:

```
private double fleschKincaidGradeLevelOf(ArrayList<String> lines)
```

This method should accept as input an `ArrayList<String>` containing the lines of a file (note that this isn't the same as a list of all the tokens in the file) and return the Flesch-Kincaid grade level score for the text of that file. As a reminder, the value should be computed as

$$Grade = C_0 + C_1\left(\frac{\text{num words}}{\text{num sentences}}\right) + C_2\left(\frac{\text{num syllables}}{\text{num words}}\right)$$

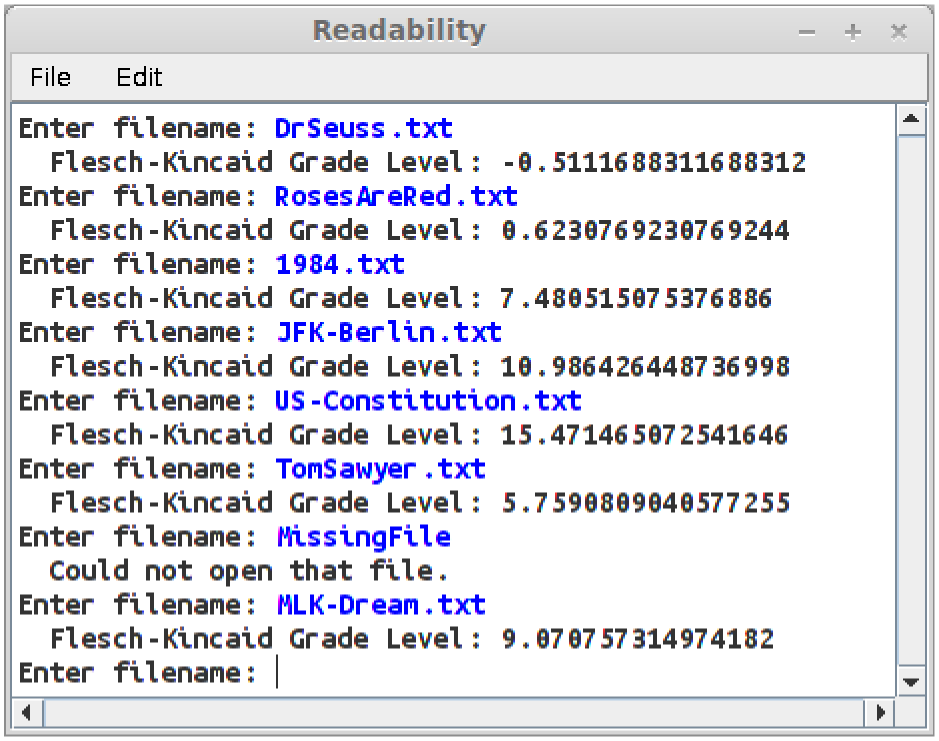
where C_0, C_1, and C_2 are these constants:

$C_0 = -15.59$ $\qquad$ $C_1 = 0.39$ $\qquad$ $C_2 = 11.8$

(I have no idea where these numbers came from, but they're the standard numbers used in computing these scores.) When computing the above formula, you should sum up the total number of words, syllables, and sentences across all the lines of the file.

You might notice that this formula isn't defined if there are zero sentences or zero total words, so in the event that the text has no words or no sentences, you should pretend that it contains a single word or a single sentence, respectively.

Having written this method, update your `run` method so that it sits in a loop prompting the user for the name of a file to read. If the user enters the name of a file, you should read that file and output its Flesch-Kincaid grade level. If the user enters an invalid filename, you should print out an error and reprompt the user. Finally, if the user hits enter without typing anything in, your program should exit. Here are some sample screenshots of our program, along with the Flesch-Kincaid grade level it reports for the different sample files we've provided:



```
Enter filename: DrSeuss.txt
  Flesch-Kincaid Grade Level: -0.5111688311688312
Enter filename: RosesAreRed.txt
  Flesch-Kincaid Grade Level: 0.6230769230769244
Enter filename: 1984.txt
  Flesch-Kincaid Grade Level: 7.480515075376886
Enter filename: JFK-Berlin.txt
  Flesch-Kincaid Grade Level: 10.986426448736998
Enter filename: US-Constitution.txt
  Flesch-Kincaid Grade Level: 15.471465072541646
Enter filename: TomSawyer.txt
  Flesch-Kincaid Grade Level: 5.759080904057725 5
Enter filename: MissingFile
  Could not open that file.
Enter filename: MLK-Dream.txt
  Flesch-Kincaid Grade Level: 9.070757314974182
Enter filename: 
```

You might notice that the `DrSeuss.txt` file gives back a negative grade level score. That's okay – the Flesch-Kincaid score can be negative for very simple texts.

Step Six: Implement The Dale-Chall Readability Score

Your job in this step is to write a method

```
        private double daleChallReadabilityScoreOf(ArrayList<String> lines)
```

As a reminder, the Dale-Chall score is computed as

$$Difficulty = D_0\left(\frac{num\ difficult\ words}{num\ words} \times D_1\right) + D_2\left(\frac{num\ words}{num\ sentences}\right) + D_3\ bonus$$

where D_0, D_1, D_2, and D_3 are chosen as

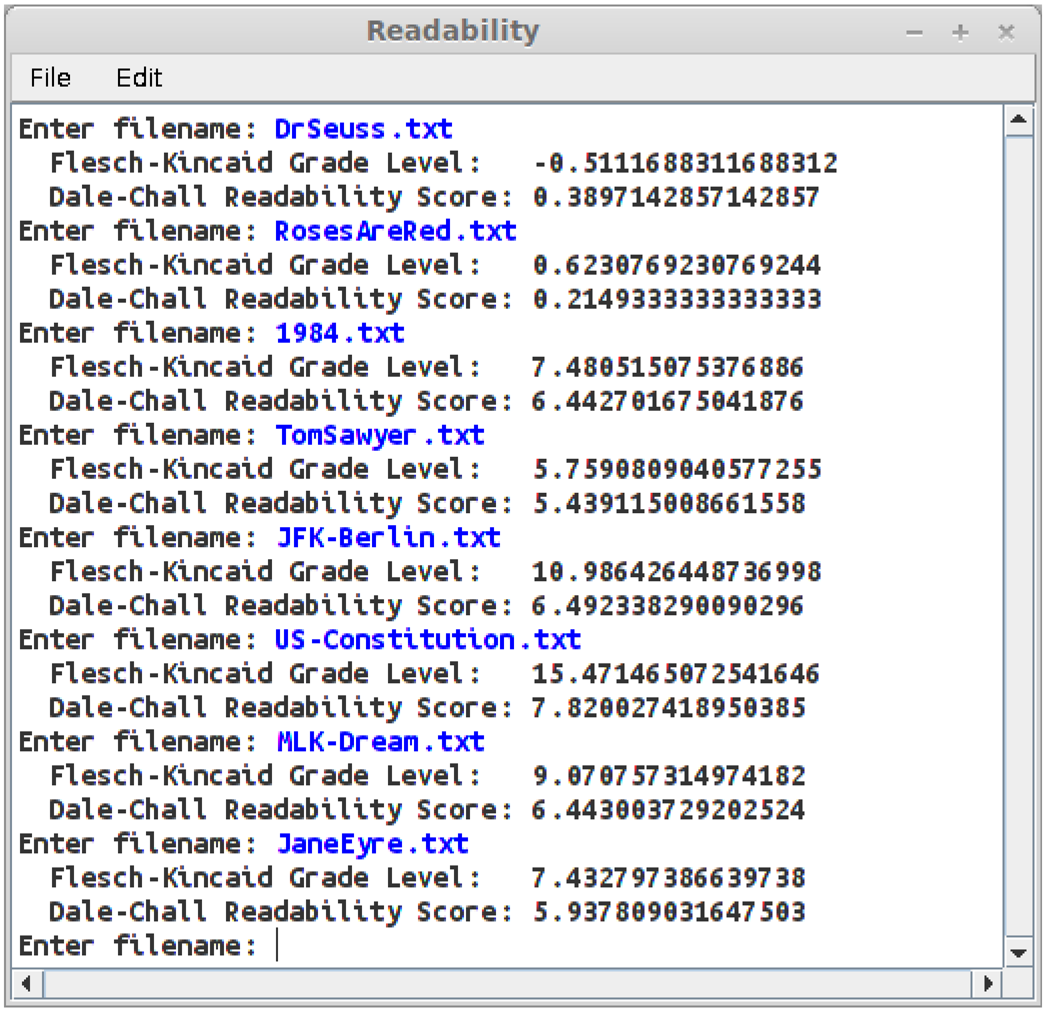
$D_0 = 0.1579$ $D_1 = 100$ $D_2 = 0.0496$ $D_3 = 3.6365$

Here, a "difficult word" is defined to be a word with three or more syllables. The *bonus* here is 1 if at least 5% of the words are difficult words and is 0 otherwise, which adds to the difficulty of the text if a large fraction of the words are difficult.

As with Flesch-Kincaid readability, if the text contains no words or no sentences, you should pretend that it has at least one word or at least one sentence, respectively.

You currently have most, but not all, of the methods you'll need to implement this method. We're going to leave the decisions about how to design and test this method up to you!

Then, update your program so that it displays both the Flesch-Kincaid grade level and the Dale-Chall readability score for the input programs. Here's a sample run of our version of the program, including the scores we computed for various pieces of text:



```
Enter filename: DrSeuss.txt
  Flesch-Kincaid Grade Level:   -0.5111688311688312
  Dale-Chall Readability Score: 0.3897142857142857
Enter filename: RosesAreRed.txt
  Flesch-Kincaid Grade Level:   0.6230769230769244
  Dale-Chall Readability Score: 0.2149333333333333
Enter filename: 1984.txt
  Flesch-Kincaid Grade Level:   7.480515075376886
  Dale-Chall Readability Score: 6.442701675041876
Enter filename: TomSawyer.txt
  Flesch-Kincaid Grade Level:   5.759080904057725
  Dale-Chall Readability Score: 5.439115008661558
Enter filename: JFK-Berlin.txt
  Flesch-Kincaid Grade Level:   10.98642644873699
  Dale-Chall Readability Score: 6.492338290090296
Enter filename: US-Constitution.txt
  Flesch-Kincaid Grade Level:   15.471465072541646
  Dale-Chall Readability Score: 7.820027418950385
Enter filename: MLK-Dream.txt
  Flesch-Kincaid Grade Level:   9.070757314974182
  Dale-Chall Readability Score: 6.443003729202524
Enter filename: JaneEyre.txt
  Flesch-Kincaid Grade Level:   7.432797386639738
  Dale-Chall Readability Score: 5.937809031647503
Enter filename: 
```

Step Seven: Interface with the Web

Wouldn't it be cool if you could also analyze the readability levels of different webpages; say, the *New York Times*, Wikipedia, or the *Stanford Daily*? In this part of the assignment, you'll do just that.

As part of the starter code, we've provided you a method you can use to get the text of a webpage as an `ArrayList<String>`. You can use it by calling

```
lines = Scraper.pageContents(url);
```

where URL is a string that starts with either `http://` or `https://`. If the page can't be opened, then this method will return `null` as a sentinel. Note that this method might take a second or two to run, since it needs to download the given webpage and extract the text of the main article on that page.

Update your program so that when the user types in the name of the document to read, your program will decide whether or not that file is a URL. If it's a URL (i.e. it starts with `http://` or `https://`), then your program should use `Scraper.pageContents` to get the contents of that page, then compute its readability. Otherwise, if it's not a URL, your program should open up a file on disk and compute its readability as before. You now have a single program that can determine the readability of any file you have on disk and any file on the web!

Step Eight: Writeup

Now that you've finished all the coding for this assignment, we'd like you to do a quick writeup answering some questions about your experience with this assignment. Edit the `writeup.txt` file with your answers to the following questions:

1. What specific words did you use to test your syllable-counting code? Why did you choose each of those words as test cases?
2. What specific strings did you use to test your `tokenize` method? Why did you choose those strings?
3. What specific strings did you use to test your code for counting syllables, words, and sentences in a line of text? Why did you choose those strings?
4. Find a piece of text of your own choosing and compute its Flesch-Kincaid grade level and Dale-Chall readability scores using the program you've written. What numbers did you get back? Were you at all surprised? (We recommend trying out your program on some text from one of your favorite books; say, one of the *Harry Potter* titles, *Fifty Shades of Grey*, or perhaps *Karel the Robot Learns Java*. Project Gutenberg, located online at http://www.gutenberg.org/, might be a good place to look for older books now in the public domain.)

Advice, Tips, and Tricks

Here are some suggestions that might be helpful as you work through this assignment:

- Remember that your heuristic for counting syllables should always report at least one syllable in each word.
- When counting syllables, remember that the words may be given with any capitalization. Your `syllablesInWord` method should produce the same outputs given as input any of the words `Capitalized`, `capitalized`, `CAPITALIZED`, or `cApItAlIzEd`.
- When implementing tokenization, we suggest avoiding the `String.split`, the `Scanner` class, and the `StringTokenizer` class (don't worry if you haven't seen these before). They are not particularly useful for splitting strings apart in the way that is required for this assignment.
- When tokenizing a string, you may need to convert a `char` into a `String`. You can do this by writing `"" + ch`, where `ch` is the character you want to convert.
- Remember that you should compare strings using the `.equals()` method and not using `==`.
- Make sure that your tokenization code works correctly on an input string that ends with a word not followed by any punctuation symbols.
- Some pieces of text will have a negative Flesch-Kincaid grade level. That's okay – it just means that the text is very simple.
- When computing the ratios in the Flesch-Kincaid and Dale-Chall readability scores, remember that integer division is not the same as real division – integer division alway rounds down, while real division won't.
- When computing readability, remember that you should treat every document as having at least one word and at least one sentence. You might get slightly different results from us if you forget to handle this or handle this improperly.

Extension Suggestions

Want to go above and beyond on this assignment? Here are some suggestions on how you can improve the program.

- The current tokenization algorithm will split words like “isn't” or “doesn't” into three pieces. Try modifying the tokenization algorithm to not split compound words like these. You might also want to adjust the syllable-counting heuristic to handle contractions elegantly.
- The “true” version of the Dale-Chall score leaves out common suffixes like “es” and “ed” from words when counting up the number of syllables they contain. Try updating your program to take this into account.
- There are many other readability indices besides Flesch-Kincaid or Dale-Chall. Choose an implement one of them as an extension.
- In a previous quarter, a student used their program to analyze the readability levels of various US patents and compared them to the average education level of a trial jury to provide an interesting window into the complexities of patent litigation. Try using your program to analyze the complexities of different pieces of text and see if you can draw any conclusions from them. If you find anything interesting and include a writeup, we'd be happy to award you extra credit.
- The readability indices described here are designed to work on written texts. Could you try to use similar heuristics to determine the complexity of materials other than written texts, such as song lyrics or poetry?

Good luck!