CS 310 Data Structures
Programming Assignment 1
Spring 2021
Due Wednesday, March 10 at midnight
Note that spring break starts Sunday, March 14

This assignment aims to help you:

- Learn more about Maps (Symbol Tables) and Sets, and get experience with their JDK classes
- Learn about string tokenization with Scanner and more specialized classes like Tokenizer
- Set up and use a Java interface
- Learn about Java packages

Reading

Read S&W Section 3.1 on symbol tables (maps), Section 3.5 on applications including text indexes like this one. If you have Weiss, read Weiss Chap. 6 through Section 6.8 on Collection classes. Weiss Chap 12, Sec. 12.2 discusses Xref.java. Read the JDK API docs on at least Map, TreeMap, TreeSet, and StringBuilder.

Steps

See Suggested Steps for more information.

Description

1. Build Xref by downloading its two sources Xref.java and Tokenizer.java from Weiss's website for this book, 4th edition. Replace "weiss" in the imports to "java", and compile the sources. Note that this program reads a Java source file and analyzes it. Run Xref on Xref.java itself, that is, use Xref.java as an input file to the Xref program, as follows:

```
java Xref < Xref.java
```

Note that Xref takes its input from standard input, so here we use redirection from an input file, as we did in pa0. Check the results by "grep lineItr Xref.java" on Linux or Mac or "find /n "lineItr" Xref.java" on Windows (in a command window), and try a few other grep/find's. Note that you can search Xref's output by grep/find: try "java Xref < Xref.java | grep lineItr" or "java Xref < Xref.java | find /n "lineItr".

2. Using Packages. See Java tutorial on packages.
Add "package cs310" to the top of the two .java files and set up a proper project with directory structure as follows:
In directory pa1, subdirs classes and src, each with cs310 subdirs for the package files. Here is output from the Window "tree" command:

```
├───bin
│   └───cs310
```

.

└──src
└──cs310

However, if you are not using an IDE, you can drop the use of that bin directory and simply have

└──src
└──cs310

With this simpler setup, build and run as follows:

```
cd src                  (the top-level source directory)
javac cs310/*.java      drops .class files in
cs310/*.class, in with .java files
java cs310.Xref < inputfile     inputfile should be a Java
source file, say cs310/Xref.java
```

If you are using an IDE, you probably see the bin directory. In this case, you can compile and run Xref by

```
cd src            (the top-level source directory)
javac -d ../bin cs310/*.java
cd ../bin         (the top-level class files directory)
java cs310.Xref < inputfile      inputfile should be a
Java source file, say ../src/cs310/Xref.java
```

The reason to be cd'd to these directories is that this is the easiest way to get the right classpath in use. The default classpath is the current directory, and you've cd'd to the right place to set the desired classpath this way. You can do the commands from other directories by specifying the classpath on the command line, like this:

`javac -cp src -bin src/cs310/*.java` (while cd'd to pa1)

`java -cp bin cs310.Xref <inputfile  inputfile is src/cs310/Xref.java for ex.`

For an IDE like eclipse, set up the directories as above, with src and bin, put the sources in src/cs310, add the package statements to *.java, and then set up a project in the IDE. Eclipse can "Open project from file system" once it is set up, allowing you to locate the project whereever you want. In memo.txt, show output of command line compile and run (using BankAccount.java as input) in this project, and `tree` output (or `du` for Linux/Mac).

3. Compose an interface called JavaTokenizer.java that expresses the public API supported by Tokenizer, and make the concrete class Tokenizer implement it. Make the client of Tokenizer, Xref.java, use the interface type once it has created such a Tokenizer object.

4. Another program needing the Java tokenization service, called Annotator.java, is supposed to input a Java source file and output the same text but with [] around each Java identifier. For this, add a method String skippedText() to the interface and Tokenizer.java. This method can be called after each time a Java identifier has been located and returned, to supply the text that the Tokenizer skipped over on the way to the identifier, or on the way to EOF (end-of-file). Note that JavaTokenizer can still be used by Xref, because there's no requirement to use all the methods of an interface in the client. Use a StringBuilder to build up the skipped text, adding chars to it one by one during

the analysis done in Tokenizer. Read the JDK documentation to see how a StringBuilder is converted to a String, for return to the client.

5. From Weiss Problem 12.20(12.17), pg. 503, the SpellChecker, simplified to have no personal dictionary, as follows:
Use a map of strings to lists of integers (like Xref's) to implement a spelling checker. Use the dictionary "words" from UNIX/Linux, available via your cs310 users.cs.umb.edu login at /usr/share/dict/words, or here. Output all misspelled words and the line numbers on which they occur, in alphabetical order by word (this is similar to Xref processing). Also, for each misspelled word, on the same line, after a colon, list any words in the dictionary that are obtainable by applying any of the following rules:

a. Add one character.
b. Remove one character.
c. Exchange adjacent characters.

Again use package cs310 (so all the pa1 source files are in the same directory.) For tokenizing the words from the input file, don't try to use Tokenizer.java (that's just for analyzing Java source code), but instead use java.util.Scanner as we did for FrequencyCounter. Try to write efficient string manipulation code for the task of checking modified strings when the user-provided string does not match in the dictionary. Don't add Strings together character by character, but rather use a StringBuilder. Assume the size of the problem is dominated by the big dictionary.
Usage: `java cs310.SpellChecker inputfile dictionary`.

memo.txt

In the file memo.txt, answer the following questions, in one to two pages (60-120 lines) of text. Use complete sentences, as you would for an English class

1. Discuss your experiences in writing these programs. What development tools (IDE, etc.) did you use? Did you develop on cs.umb.edu servers, or if on your own PC/Mac, did you have any problem recompiling and running on Linux?
2. As described in problem 2 above, command line compile and run, display of directories by tree or du.
3. Estimate the maximum memory in use for N input lines being tokenized by Tokenizer. Note that a Java character uses 2 bytes.
4. Analyze the big-O CPU performance of the spell-checker, for N words in the big dictionary and O(1) words to check.

Delivery

Before the due date, assemble files in the pa1 subdirectory of your provided cs310 directory on our UNIX site. Make sure they compile and run on Linux! They should, since Java is very portable. It's mainly a test that the file transfer worked OK. Note that all the sources should be in the cs310 package, and thus are located in the src/cs310 directory.

- `pa1/memo.txt` (plain txt file, try “more memo.txt” on users.cs.umb.edu)
- `pa1/src/cs310/Xref.java (provided, but needs small edits)`
- `pa1/src/cs310/JavaTokenizer.java (an interface)`
- `pa1/src/cs310/Tokenizer.java (provided, but needs edits)`
- `pa1/src/cs310/Annotator.java`
- `pa1/src/cs310/SpellChecker.java`

  Check your filenames: login on cs.umb.edu Linux: cd cs310/pa1, then “ls” should show memo.txt, src, bin. “ls src/cs310” should show all the source filenames. Remember that case counts on UNIX!