CS 310 Data Structures
Programming Assignment 1
Spring 2021

Suggested Steps, more discussion.

1. Xref and Tokenizer. Make a pa1 directory and a src subdirectory, next to your pa0 directory. Download Xref.java and Tokenizer.java from Weiss to your src directory, i.e. pa1/src relative to the top level. Build and run the supplied programs (without a package) in the src directory.

```
javac *.java
java Xref < Xref.java (or another Java source file put in this directory)
```

2. **Using package cs310.** Java programmers use packages to help prevent name collisions in software, and help organize code. Google "Java name collision" for more info. As soon as a project uses more than a handful of files, it's good to use packages. To get started, we'll just use one named package here for the application's code. Each package is required to have its own directory under the source-file area, src in our case. Our package is named cs310, so cd to src and mkdir cs310 to create the needed directory for source files in package cs310.

Once you start using a package, you should stop using the "default package" that has no name (for parts of the same program). They don't mix well as parts of a program, at least for calling from a named package into the default package. To move Xref/Tokenizer to the cs310 package, copy the two files into src/cs310 and add "package cs310;" to the top of each file. Then compile as shown in the assignment. The package specification does not affect how a single-file code executes, if it executes at all.

Use the compile and run guide in the assignment to run Xref with the code in a package.

Note that Xref.java has an overly-complicated code in its main() method. In general, main (being static) is not considered part of the object API, but rather as client code that happens to be inside the object's source file. So if you prefer, simplify main to:

```
 public static void main( String [ ] args )
{
    Xref p;
    System.out.println("Starting...");
    if( args.length == 0 )
    {
        p = new Xref( new Tokenizer(new InputStreamReader( System.in ) ));
        p.generateCrossReference( );

        return;
    } else System.out.println("Xref usage: java Xref < inputfile");
 }
```

Part 3. Tokenizers. See Wikipedia on tokenizers, under lexical analyzers, a part of parsing. To emphasize the API between the parts of this program, we define a Java interface between Xref and Tokenizer. Note that understanding how and when to use interfaces is an important topic of this course.

Here the JavaTokenizer interface will describe the actions (methods) that the tokenizer can do, and Xref will call those methods by calling the methods of the interface (instead of directly calling the concrete class the interface is describing). We call Xref the "client" of the interface and the tokenization service it describes. The interface describes the API of the tokenization service, or more exactly the methods of the API, since the full API should include a way to create the service. The service creation step is necessarily specific to the concrete class of the

service (it's of form ... = new ConcreteClass(...)), so you see that an interface does what can be done to make the specification valid across implementations.

Suggested steps:

1. Figure out the public API of the Tokenizer, i.e., the set of public methods implemented.

2. Write the interface file, JavaTokenizer.java. Add "implements JavaTokenizer" to the tokenizer.

3. Find the code in Xref.java that creates a tokenizer. You see it has the name of the concrete class written into the code. Now that we have an interface, we can make use of it. The Xref code only needs the interface methods, so will be happy with the interface API. But it needs an *object* with that API, so you might think it needs a "new SomeClass()". But there's another way. We can arrange to pass an already-instantiated object into the Xref constructor. Then we're shifting the responsibility of creating the object to the client of Xref. This is considered better software engineering. To do this step, simply make the Xref constructor take a JavaTokenizer argument instead of the old Reader argument, and save it in the same tok instance variable as before, now made to be type JavaTokenizer. In main, which is considered client code of Xref, create a Tokenizer and feed it to the Xref constructor.

4. Build the program. The build commands are in the assignment for part 2.

5. Run the program: the output should be very much like the first run.

Part 4. Adding to the interface.

The Annotation program needs both the id strings and the text between the id strings in the given file. Here is a little example of its output: each Java id has been surrounded by square brackets. Any square brackets already in the program are preserved.

```
// a little class for testing
[public] [class] [Box] {
   // count of something
   [private] [int] [count];
   /* inc method */
   [public] [void] [increment]() {
      [count]++;  // line comment
    }
    [public] [int] [getCount]() { [return] [count] }
 }
```

You can see the original program if you drop all the square brackets. So we need another method that returns the skipped-over text as the Tokenizer searches for the next Java id. It's another action like the ones described in the interface, so it's reasonable to add a method to that interface.

1. Add the new method to the JavaTokenizer interface and a stub method (doesn't do the real work yet) to Tokenizer.java

2. Compile to check syntax.

3. Write Annotator.java. It's very simple. It just loops through calls to the tokenizer's getNextId(), and for each, calls skippedText(), printing the skipped text, then print [, print the id, print].

4. (edited 3/3) Implement the new method in Tokenizer. You need a new instance variable of type StringBuilder to hold the growing skipped-text string efficiently. To add a char ch to it, use .append(ch). Go through the code of Tokenizer, studying the cases of reading in a char, deciding whether or not it's in a Java id or not, and if it is processed and is not part of an id, append it to your skipped-text StringBuilder. This skipped-text StringBuilder

grows during the execution of getNextId, and it ready to deliver once getNextId is finished. Note that sometimes in Tokenizer the incoming char is "pushed back" into the input stream--that's a case where it's not processed, so don't add it to the StringBuilder in that case. Test with Annotator.

SpellChecker

Note there are several possible ways to set this up. You don't have to follow this one way. But do use object methods, except of course main(), the needed static method to get the program started.

1. This needs two major data structures: the big map to store the misspelled words and their line numbers, and the dictionary of well-spelled words, which is just a set of strings. Set up these as instance variables. (edited 3/3) You also need to find a decent input file, a text file with both properly-spelled words and misspelled words.

2. First ignore the substitution rules and just find the misspelled words. Also ignore the big map for now. The constructor can accept the dictionary filename and load its data into the set of strings. For this we need a Scanner for a file specified by its filename (String). There are several ways to do this. See StackOverflow 309021073. The simplest way is new Scanner(new File(filename)), and this discussion explains that as long as you close the Scanner, the OS file will be properly closed.

3. Then we need the method that processes a certain file for spelling and takes that filename as an argument. Again we need a Scanner based on a filename to tokenize it into words. Check each word against the dictionary. This version reports the misspelled words as they occur.

4. Now add line numbers and the big map into the processing. For line numbers, just use the scanner to read in whole lines at a time, increment the line number, and String.split them up. When you find a misspelled word, get its list out of the big map and add the current line number. Note that you don't have to map.put the list back: you've been modifying the actual list in the big map.

5. Finally, use a StringBuilder to modify the individual word-strings as specified and check them against the dictionary.