ICSI 311 Assignment 1 – The Lexer

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. Especially ".class" files.

You must not zip or otherwise compress your assignment. Blackboard will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

Introduction

In this project, we will begin our lexer. Our lexer will start by reading the strings of the BASIC file that the user wants to run. It will break the BASIC code up into "words" or tokens and build a collection of these tokens. We can consider the lexer complete when it can take any BASIC file and output a list of the tokens being generated.

We will not be using the Scanner class that you may be familiar with for reading from a file; we are, instead, using Files.readAllBytes. This is a much simpler way of dealing with files.

Example of readAllLines:

```
Path myPath = Paths.get("someFile.basic");
String content = new String(Files.readAllBytes (myPath);
```

A second concept you may not be familiar with is "enum". Enum, short for enumeration. This is a Java language construct that lets us create a variable that may be any one of a list of things. We will use this to define the types of tokens – a tokenType.

```
enum colorType { RED, GREEN, BLUE }
colorType myFavorite = colorType.BLUE;
System.out.println(myFavorite); // prints BLUE
```

The general way to think about a lexer is the same way that you learned to read. Remember following the line of letters with your finger? We will do the same thing – we will create a class (I called mine "CodeHandler"). This class will "manage" the incoming stream of letters, allowing the lexer to "peek" ahead in the stream, get a character (move the finger one forward), and tell us if we are at the end of the document.

With the CodeHandler written, we will move on to the lexer itself. For now, we will only deal with a few types of tokens – words, numbers and new line. Of course, just "reading" through the document doesn't do anything – we will make tokens – objects that hold the type (using an enum) and a string of the value. For example:

WORD(hello)

Details

This assignment **must** have four different source code files.

Basic.Java

Basic.java **must** contain main. Your main **must** ensure that there is one and only one argument (args). If there are none or more than 1, it **must** print an appropriate error message and exit. That one argument will be considered as a filename. You will pass the filename to an instance of Lexer (which we will build below). You will then call lex() on your lexer. It will return a linked list of Token. Print each token out. This is "just" debugging output – the format isn't super important so long as it makes sense to a programmer.

CodeHandler

The code handler class should have a private string to hold the document (BASIC file) and a private integer index (the finger position). It should have methods:

char Peek(i) -looks "i" characters ahead and returns that character; doesn't move the index
String PeekString(i) – returns a string of the next "i" characters but doesn't move the index
char GetChar() – returns the next character and moves the index
void Swallow(i) – moves the index ahead "i" positions
boolean IsDone() – returns true if we are at the end of the document
String Remainder() – returns the rest of the document as a string
Some of these are used in next assignment. Note that there is no accessor for the string array. You **must** use the methods.
The constructor will take a filename. Basic will call Lexer which will call CodeHandler. Use ReadAllBytes to read from the file and populate the private internal string.

Token

Create the Token class. It needs an enum of TokenType (values: WORD, NUMBER, ENDOFLINE) and a string to hold the value of the token (for example, "hello" and "goodbye" are both WORD, but with different values. The token will also hold the line number and character position of the start of the token. Create two constructors – one for TokenType, line number and position and one that also has a value; some tokens don't have a value because it is doesn't matter (new line). Make sure to add a ToString method. Your exact format isn't critical; I output the token type and the value in parentheses if it is set.

Lexer

The final file **must** be called Lexer.java. The Lexer class **must** contain a lex method that accepts a single string (the filename) and returns a linked list of Tokens. Your lexer must keep track of the line number and character position within the line. This will help you generate good error messages.

Lex is the method that will break the data from CodeHandler into a linked list of tokens. While there is still data in CodeHandler, we want to peek at the next character to get an idea what to do with it.

- If the character is a space or tab, we will just move past it (increment position).
- If the character is a linefeed (\n), we will create a new EndOfLine token with no "value" and add it to token list. We should also increment the line number and set line position to 0.
- If the character is a carriage return (\r), we will ignore it.
- If the character is a letter, we need to call ProcessWord (see below) and add the result to our list of tokens.
- If the character is a digit, we need to call ProcessDigit (see below) and add the result to our list of tokens.
- Throw an exception if you encounter a character you don't recognize.

ProcessWord is a method that returns a Token. It accepts letters, digits and underscores (_) and make a String of them. When it encounters a character that is NOT one of those, it stops and makes a “WORD” token, setting the value to be the String it has accumulated. Make sure that you are using “peek” so that whatever character does NOT belong to the WORD stays in the StringHandler class. Also remember to increment the position value. Words can also end with $ and %.

ProcessNumber is similar to ProcessWord, but accepts 0-9 and one “.”.

Create Junit tests for your lexer **and** for CodeHandler.

Test with multi-line strings. Test with words then numbers and numbers and then words.

Some examples of valid input and the result output are:

Input	***Output***
an empty line	EndOfLine
5 hello	NUMBER (5) WORD(hello) EndOfLine
5.23 8.5 3	NUMBER(5.23) NUMBER(8.5) NUMBER(3) EndOfLine
8 4 99999	NUMBER (8) NUMBER (4) NUMBER (99999) EndOfLine
7 4 3 1	NUMBER(7) NUMBER(4) NUMBER(3) NUMBER(1) EndOfLine
2 number 3	NUMBER(2) WORD (number) NUMBER (3) EndOfLine

HINTS

Do not wait until the assignment is nearly due to begin. Start early so that you can ask questions.

You may find it useful to make some private helper methods in your lexer. **This is encouraged.**
Read the rubric carefully.
Be careful about good OOP – very little in this assignment should be “static”. Members should always be private.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
Unit Tests	Don't exist (0)	At least one (3)	Missing tests (6)	All functionality tested (10)
Basic.java/main	Doesn't exist or named wrong (0)			Exists and named correctly (5)
Basic - Calling lex	Non-Existent (0)	Reuses the lexer or doesn't call lex appropriately(3)		Instantiates Lexer and calls lex for each line (5)
Handling exceptions	Don't handle exception (0)			Handles exception and prints message(5)
Printing results	Doesn't print results			Prints results appropriately(5)
Code Handler	Doesn't exist (0)	Exists and holds string and index (3)	Exists, members correct, constructor correct (6)	All methods, members, constructor correct (10)
Code Handler - File reading	Non-Existent (0)	Uses some other mechanism for reading lines from a file(2)		Uses RealAllBytes(5)
Token Class	Non-Existent (0)			Has enum, integers for line/position, string (5)
Lexer – constructor	Doesn't exist (0)		Instantiates StringHandler (3)	Instantiates CodeHandler and sets line number and position (5)
Lexer – lex	Doesn't exist (0)	Exists and loops over the string (3)	Exists, loops over the string, returns a LinkedList of tokens (3)	Skips appropriate values, calls ProcessWord and ProcessNumber and adds their return values to the list (5)
Lexer – ProcessWord	Doesn't exist (0)	One of: Accepts required characters, creates a token, doesn't accept characters it shouldn't (3)	Two of: Accepts required characters, creates a token, doesn't accept characters it shouldn't (6)	Accepts required characters, creates a token, doesn't accept characters it shouldn't (10)
Lexer – ProcessNumber	Doesn't exist (0)	One of: Accepts required characters, creates a token, doesn't accept characters it shouldn't (3)	Two of: Accepts required characters, creates a token, doesn't accept characters it shouldn't (6)	Accepts required characters, creates a token, doesn't accept characters it shouldn't (10)
Lexer – tracks line/position	Doesn't (0)			Does and populates each token (5)