10/14/2021

Faith Swetnam

Kennesaw state university

Faith Swetnam

Fall 2021

CS 4308/Section 03

1st Deliverable – Scanner

Professor Sharon Perry

100% complete and working as designed

Table of Contents

[Source Code 2](#_Toc85210260)

[Lexical\_Analyzer.java 2](#_Toc85210261)

[Token.java 9](#_Toc85210262)

[Error.java 11](#_Toc85210263)

[Code Overview 11](#_Toc85210264)

[LexicalAnalyzer.java 11](#_Toc85210265)

[Variables 11](#_Toc85210266)

[Methods 12](#_Toc85210267)

[Token.java 15](#_Toc85210268)

[Error.java 15](#_Toc85210269)

[Screenshots (Successful runs and some errors) 16](#_Toc85210270)

[Test1.jl Output 16](#_Toc85210271)

[Test2.jl Output 17](#_Toc85210272)

[Test3.jl 18](#_Toc85210273)

[File Not Found Error 19](#_Toc85210274)

[Lexeme is too long 20](#_Toc85210275)

[No tokens initialized error 21](#_Toc85210276)

[Program ended unexpectedly part 1 22](#_Toc85210277)

[Program ended unexpectedly part 2 23](#_Toc85210278)

[Unexpected symbol 24](#_Toc85210279)

[References 24](#_Toc85210280)

# Source Code

## Lexical\_Analyzer.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 1 Scanner  
 \*/  
  
import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.ArrayList;  
import java.util.Scanner;  
  
public class LexicalAnalyzer {  
  
 //Global static variables  
 static char *nextChar*; //char to store the next character from file  
 static char[] *sourceArray* = new char[100]; //stores all characters from the source file  
 static ArrayList<Character> *lexeme* = new ArrayList<Character>(); //stores lexemes  
 static int *sourceCount* = 0; //stores position in sourceArray array  
 static Token.CharacterClass *currCharClass*; //CharacterClass of nextChar  
 static Token.CharacterClass *prevCharClass*; //CharacterClass for previous nextChar  
 static ArrayList<Token> *tokens* = new ArrayList<Token>(); //stores all tokens produced  
 static int *sourceLine* = 1; //stores the line of source the lexeme is on  
 static ArrayList<Error> *errors* = new ArrayList<Error>(); //stores errors that are found  
 static boolean *errorOccurred* = false; //stores whether an error has occurred  
 static String *validSymbols* = "=<>~+-/\*\_()"; //stores valid symbols for Julia  
  
 //ReadFile reads File f and returns a char[] of contents of the file  
 //Adds a '\n' at the end of each line and '\u001a' at the end of the file for processing reasons  
 static char[] readFile(File f) {  
 String source = "";  
 try {  
 Scanner fileReader = new Scanner(f);  
 while(fileReader.hasNextLine()) {  
 source += fileReader.nextLine() + '\n';  
 }  
 source += '\u001a';  
 *sourceArray* = source.toCharArray();  
 fileReader.close();  
 return *sourceArray*;  
 } catch (FileNotFoundException e) {  
 Error err = new Error("File could not be found");  
 *errors*.add(err);  
 *errorOccurred* = true;  
 return *sourceArray*;  
 }  
 }  
  
 //GetChar returns the next character from sourceArray  
 //It sets the currCharClass based on value of nextChar  
 static char getChar() {  
 *nextChar* = *sourceArray*[*sourceCount*];  
 if(*nextChar* != '\u001a') {  
 if(Character.*isLetter*(*nextChar*)) {  
 *currCharClass* = Token.CharacterClass.*LETTER*;  
 } else if (Character.*isDigit*(*nextChar*)) {  
 *currCharClass* = Token.CharacterClass.*DIGIT*;  
 } else {  
 *currCharClass* = Token.CharacterClass.*UNKNOWN*;  
 }  
 *sourceCount*++;  
 } else {  
 *currCharClass* = Token.CharacterClass.*EOF*;  
 }  
 return *nextChar*;  
 }  
  
 //AddChar adds the value in nextChar to lexeme  
 //if lexeme is too long it logs an error  
 static void addChar() {  
 if(*lexeme*.size() < 100) {  
 *lexeme*.add(*nextChar*);  
 } else {  
 Error err = new Error("value is too long", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 }  
 }  
  
 //Function to process (skip over) white-spaces, tabs, new lines in the file  
 //Logs an error if program ends unexpectedly  
 static void processFormatting() {  
 while (Character.*isSpaceChar*(*nextChar*) || *nextChar* == '\t' || *nextChar* == '\n') {  
 if(*nextChar* == '\n'){  
 *sourceLine*++;  
 }  
 if(*nextChar* != '\u001a') {  
 *getChar*();  
 } else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 }  
 }  
  
 }  
  
 //Function to process single line comments  
 //Logs an error if program ends during comment  
 static void comments() {  
 if(*nextChar* == '/' && *sourceArray*[*sourceCount*] =='/') {  
 *processFormatting*();  
 while(*nextChar* != '\n') {  
 if(*nextChar* != '\u001a') {  
 *getChar*();  
 } else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 }  
 }  
 }  
 }  
  
 //String turns the lexeme into a string for processing purposes  
 //It returns the string lex when finished  
 static String string() {  
 String lex = "";  
  
 if(*lexeme*.size() > 1) {  
 StringBuilder builder = new StringBuilder(*lexeme*.size());  
 for(Character c: *lexeme*) {  
 if(c.charValue() != '\u001a')  
 builder.append(c);  
 else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 break;  
 }  
 }  
 lex = builder.toString();  
 } else if (*lexeme*.size() > 0){  
 if(*lexeme*.get(0) != '\u001a')  
 lex = Character.*toString*(*lexeme*.get(0));  
 else {  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 }  
 } else {  
 Error err = new Error("lexeme array is empty", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 }  
  
 return lex;  
 }  
  
 //adds token to token arraylist  
 static void addToken(Token.TokenType tokenType, String lex, int sourceLine){  
 Token currToken = new Token(tokenType, lex, sourceLine);  
 *tokens*.add(currToken);  
 }  
  
 //LookUp determines the token type based on the lexeme  
 static void lookUp(String lex) {  
 Token.TokenType tokenType;  
 //if the lexeme is a word(contains alphabetical letters) do  
 if(*prevCharClass* == Token.CharacterClass.*LETTER*) {  
 switch (lex) {  
 case "end":  
 tokenType = Token.TokenType.*EOF*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "function":  
 tokenType = Token.TokenType.*FUNCT*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "while":  
 tokenType = Token.TokenType.*WHILE*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "do":  
 tokenType = Token.TokenType.*DO*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "print":  
 tokenType = Token.TokenType.*PRINT*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "if":  
 tokenType = Token.TokenType.*IF*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "then":  
 tokenType = Token.TokenType.*THEN*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "else":  
 tokenType = Token.TokenType.*ELSE*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 default: //lexeme is an identifier  
 tokenType = Token.TokenType.*LETTER*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 }  
 //else lexeme is a symbol or number  
 } else {  
 switch(lex) {  
 case "=":  
 tokenType = Token.TokenType.*ASSIGN\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "<=":  
 tokenType = Token.TokenType.*LE\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "<":  
 tokenType = Token.TokenType.*LT\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case ">=":  
 tokenType = Token.TokenType.*GE\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case ">":  
 tokenType = Token.TokenType.*GT\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "==":  
 tokenType = Token.TokenType.*EQ\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "~=":  
 tokenType = Token.TokenType.*NE\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "+=":  
 tokenType = Token.TokenType.*AE\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "+":  
 tokenType = Token.TokenType.*ADD\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "-":  
 tokenType = Token.TokenType.*SUB\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "\*":  
 tokenType = Token.TokenType.*MUL\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "/":  
 tokenType = Token.TokenType.*DIV\_OP*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case "(":  
 tokenType = Token.TokenType.*L\_PAREN*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 case ")":  
 tokenType = Token.TokenType.*R\_PAREN*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 default:  
 tokenType = Token.TokenType.*DIGIT*;  
 *addToken*(tokenType, lex, *sourceLine*);  
 break;  
 }  
 }  
 }  
  
 //checkValid determines if the unknown symbol is valid in the language  
 static void checkValid(){  
 String check = Character.*toString*(*nextChar*);  
 if(!Character.*isLetterOrDigit*(*nextChar*)) {  
 if (!*validSymbols*.contains(check)) {  
 Error err = new Error("unexpected symbol", check, *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 }  
 }  
 }  
  
 //main body of the lexical\_analyzer  
 //returns a token based on the lexeme found  
 static void lexer() {  
 *lexeme* = new ArrayList<Character>();  
 *comments*();  
 *processFormatting*();  
 switch (*currCharClass*) {  
 case *LETTER*:  
 *addChar*();  
 *getChar*();  
 while((*currCharClass* == Token.CharacterClass.*LETTER* || *nextChar* == '\_') && !*errorOccurred*) {  
 *addChar*();  
 *getChar*();  
 }  
 *prevCharClass* = Token.CharacterClass.*LETTER*;  
 break;  
 case *DIGIT*:  
 *addChar*();  
 *getChar*();  
 while((*currCharClass* == Token.CharacterClass.*DIGIT* || *nextChar* == '.') && !*errorOccurred*) {  
 *addChar*();  
 *getChar*();  
 }  
 *prevCharClass* = Token.CharacterClass.*DIGIT*;  
 break;  
 case *UNKNOWN*:  
 *comments*();  
 *processFormatting*();  
 *checkValid*();  
 *addChar*();  
 *getChar*();  
 while(*validSymbols*.contains(Character.*toString*(*nextChar*)) && *nextChar* != '(' && *nextChar* != ')' && !*errorOccurred*){  
 *addChar*();  
 *getChar*();  
 }  
 *prevCharClass* = Token.CharacterClass.*UNKNOWN*;  
 break;  
 default:  
 *prevCharClass* = Token.CharacterClass.*EOF*;  
 break;  
 }  
 if (*prevCharClass* != Token.CharacterClass.*EOF* && !*errorOccurred*) {  
 String lex = *string*();  
 *lookUp*(lex);  
 }  
  
 }  
  
 //checks to ensure that the program ended correctly with 'end'  
 //returns a boolean  
 static boolean endedCorrectly(){  
 if(*tokens*.size() == 0){  
 Error err = new Error("no tokens initialized", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 } else if(*tokens*.get(*tokens*.size()-1).type != Token.TokenType.*EOF*){  
 Error err = new Error("program ended unexpectedly", *sourceLine*);  
 *errors*.add(err);  
 *errorOccurred* = true;  
 }  
 return *errorOccurred*;  
 }  
  
 //printTokenTable takes in an ArrayList of Token objects and prints out a symbol table  
 static void printTokenTable(ArrayList<Token> tokens) {  
 System.*out*.printf("%-10s\t%-20s\t%-10s\t%-10s\n", "Lexeme", "Token Type", "Opcode", "Line");  
 System.*out*.println("-----------------------------------------------------\n");  
 for(Token t: tokens) {  
 t.printToken();  
 }  
 }  
  
 //printErrorTable takes in an ArrayList of Error objects and prints them out in a table  
 static void printErrorTable(ArrayList<Error> errors) {  
 System.*out*.printf("%-50s\t%-10s\t%-10s\n", "Errors", "Value", "Line Occured");  
 System.*out*.println("-----------------------------------------------------------------------------\n");  
 for(Error e: errors) {  
 e.printError();  
 }  
 }  
  
  
 public static void main(String args[]) {  
 File f = new File("src/Julia-Files/Test1.jl");  
 *readFile*(f);  
 *getChar*();  
 while(*currCharClass* != Token.CharacterClass.*EOF* && !*errorOccurred*) {  
 *lexer*();  
 }  
  
 *endedCorrectly*();  
  
 if(*errorOccurred*)  
 *printErrorTable*(*errors*);  
 else  
 *printTokenTable*(*tokens*);  
  
 }  
  
}

## Token.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 1 Scanner  
 \*/  
  
//Token object holds a tokens TokenType type, lexeme (value), and the line its on  
public class Token {  
 TokenType type;  
 String lexeme;  
 int line;  
  
 //Token constructor  
 Token(TokenType type, String lexeme, int line){  
 this.type = type;  
 this.lexeme = lexeme;  
 this.line = line;  
 }  
  
 //Prints out token values  
 void printToken() {  
 System.*out*.printf("%-10s\t%-20s\t%-10d\t%-10d\n", lexeme, type.label, type.opcode, line);  
 }  
  
 //Enumerated type to hold the type of character in nextChar  
 //Each CharacterClass has a description for printing  
 enum CharacterClass {  
 *LETTER* ("identifier"),  
 *DIGIT* ("number"),  
 *UNKNOWN* ("unknown symbol"),  
 *EOF*("end of program");  
  
 final String description;  
  
 private CharacterClass(String description) {  
 this.description = description;  
 }  
 }  
  
 //Enumerated type to hold legal keywords and symbols (essentially the legal token types allowed by language)  
 //Each TokenType has an associated label and opcode. It should contain the String value that holds the lexeme  
 enum TokenType {  
 *LETTER*("identifier", "", 1),  
 *DIGIT*("number", "", 2),  
 *EOF*("keyword\_end", "end", 99),  
 *FUNCT*("keyword\_function", "function", 1001),  
 *WHILE*("keyword\_while", "while", 1002),  
 *DO*("keyword\_do", "do", 1003),  
 *PRINT*("keyword\_print", "print", 1004),  
 *IF*("keyword\_if", "if", 1005),  
 *THEN*("keyword\_then", "then", 1006),  
 *ELSE*("keyword\_else", "else", 1007),  
 *ASSIGN\_OP*("assignment\_operator", "=", 2000),  
 *LE\_OP*("less\_equal", "<=", 2001),  
 *LT\_OP*("less", "<", 2002),  
 *GE\_OP*("greater\_equal", ">=", 2003),  
 *GT\_OP*("greater", ">", 2004),  
 *EQ\_OP*("equal", "==", 2005),  
 *NE\_OP*("not\_equal", "~=", 2006),  
 *AE\_OP*("addition\_assignment", "+=", 2007),  
 *ADD\_OP*("addition\_operator", "+", 2008),  
 *SUB\_OP*("subtraction\_operator", "-", 2009),  
 *MUL\_OP*("multiplication\_operator", "\*", 2010),  
 *DIV\_OP*("division\_operator", "/", 2011),  
 *L\_PAREN*("left\_parenthesis", "(", 2012),  
 *R\_PAREN*("right\_parenthesis", ")", 2013);  
  
 final String label;  
 final int opcode;  
 String value;  
  
 //TokenType constructor  
 private TokenType(String label, String value, int opcode) {  
 this.label = label;  
 this.value = value;  
 this.opcode = opcode;  
 }  
 }  
}

## Error.java

/\*  
Class: CS 4308 Section 03  
Term: Fall 2021  
Name: Faith Swetnam  
Instructor: Sharon Perry  
Project: Deliverable 1 Scanner  
 \*/  
  
//Class to store error objects  
//An error object stores an error message, the value that caused the error (if applicable) and the line the error  
//occurred on  
public class Error {  
 String msg = "";  
 String value = null;  
 int line = 0;  
  
 //Error object if value is available  
 Error(String msg, String value, int line){  
 this.msg = msg;  
 this.value = value;  
 this.line = line;  
 }  
  
 //Error object if value unavailable  
 Error(String msg, int line){  
 this.msg = msg;  
 this.line = line;  
 }  
  
 //Error object if no value or line if specified  
 Error(String msg){  
 this.msg = msg;  
 }  
  
 //Prints the error object values out  
 void printError(){  
 System.*out*.printf("%-50s\t%-10s\t%-10d\n",msg, value, line);  
 }  
}

# Code Overview

## LexicalAnalyzer.java

### Variables

The lexical analyzer has 11 static variables.

* nextChar
  + A character variable that stores the current character being evaluated.
* sourceArray
  + A character array that holds all the characters from the source file.
* sourceCount
  + An integer that stores the programs current position in sourceArray.
* sourceLine
  + An integer that stores from which line in the source file nextChar comes from.
* Lexeme
  + An ArrayList of Character objects. This stores all the characters of a lexeme.
* currCharClass
  + A CharacterClass variable that stores the CharacterClass of the character in nextChar
* prevCharClass
  + A CharacterClass variable that stores the CharacterClass of the previous nextChar value
* tokens
  + An ArrayList of Token objects. It stores all the tokens created for the source code.
* Errors
  + An ArrayList of Error objects. It stores all the errors generated by the source code.
* errorOccurred
  + A boolean variable that is true if an error has occurred and false otherwise.
* validSymbols
  + A String variable that holds legal symbols for Julia.

### Methods

The LexicalAnalyzer class has 13 methods.

* readFile(File f)
  + readFile() takes in a Java File object and reads it into a String variable called source using a Scanner object. After each line ‘\n’ is added and at the end of the file it adds ‘\u001a’, for processing reasons. Source is then turned into a character array and stored in sourceArray. If there is a problem with opening the file, an Error object is created (“file could not be found”).
* getChar()
  + nextChar is changed to the next character in sourceArray and then its CharacterClass is determined. If the character is ‘\u001a’, the file has ended and currCharClass is set to CharacterClass.EOF. If the character is a letter or number, currCharClass is set to CharacterClass.LETTER or CharacterClass.DIGIT respectively. Otherwise, the CharacterClass is UNKNOWN.
* addChar()
  + addChar adds the value in nextChar to lexeme. If the lexeme is longer than 100 characters, an Error object is created (“value is too long”).
* processFormatting()
  + processFormatting() skips over all of the formatting in the source file. While nextChar is a whitespace, tab, or new line, it calls getChar(). It creates an Error object if the program ends unexpectedly (“program ended unexpectedly” (‘\u001a’)). If a character is a new line, it increments sourceLine, because the program has moved to the next line from the source code.
* comments()
  + This method processes out single line comments. If two backslashes appear, the method calls getChar() until a new line is reached. An Error object is created if the program ends (“program ended unexpectedly”( ‘\u001a’)).
* string()
  + string() creates a string called lex that stores all the characters from the lexeme array.
  + If lexeme has more than 1 variable, the method uses StringBuilder make a string from the characters in lexeme. If lexeme has only 1 element, the character is parsed into a String and stored in lex. Otherwise, lexeme is empty, and an Error object is created (“lexeme array is empty”).
  + If the program ends during the method an Error object is created (“program ended unexpectedly” (‘\u001a’)).
* addToken(Token.TokenType tokenType, String lex, int sourceLine)
  + This method creates a Token object using its passed values and adds it into the token array.
* lookUp(String lex)
  + lookUp() takes in the String lex, which is a string representing the characters in the lexeme array. It uses two switch statements to compare the string to keywords/reserved words and operators of the language. It uses an if-else block to speed up execution.
  + If the prevCharClass is equal to CharacterClass.LETTER, it checks for keywords/reserved words. The default case is initializes the token with a TokenType of TokenType.LETTER, which means it is an identifier.
  + Otherwise, another switch statement checks lex against valid operators. The default case for this switch statement, initializes the token with a TokenType of TokenType.DIGIT, meaning the lexeme is a number.
  + For example, if lex = “do”, case “do” executes. A TokenType called tokenType is initiated as TokenType.DO. addToken() is called to add a Token to token with tokenType, lex, and the current sourceLine as values.
* checkValid
  + This method checks if the symbol in nextChar is valid. Otherwise, an Error object is created (“unexpected symbol”)
* lexer()
  + This is the main method of LexicalAnalyzer. First, it clears lexeme. Then it calls comments() and processFormatting(), to clean up any comments and formatting. Then the switch statement begins. It has 4 cases:
    - LETTER

This case first calls addChar() and getChar(). Then, as long as an error has not occurred, and the currCharClass equals CharacterClass.LETTER or nextChar equals underscore it calls addChar() and getChar(). Essentially this builds words like ‘function’ and possible variable names like “capital\_city”. Then prevCharClass is set to CharacterClass.LETTER and the case ends.

* + - DIGIT

This case first calls addChar() and getChar(). Then, as long as an error has not occurred, and the currCharClass equal CharacterClass.DIGIT or nextChar equals ‘.’, a while loop calls addChar() and getChar(). This case can create numbers like “1234” or “12.4”. The prevCharClass is changed to CharacterClass.DIGIT and the case ends.

* + - UNKNOWN

First this case checks if the UNKNOWN character is because of formatting with processFormatting() or a comment with comments(). It checks if the character is a valid symbol in the language using checkValid(). Then it calls addChar() and getChar(). After the function calls, a while loop initializes. While an error has not occurred, and nextChar contains a valid symbol and nextChar is not a parenthesis, addChar() and getChar() are called. Then the prevCharClass is set equal to CharacterClass.EOF.

* + - Default case

This case sets prevCharClass equal to CharacterClass.EOF and breaks.

After the switch statement ends an if statement starts. If prevCharClass does not equal CharacterClass.EOF and an error has not occurred, string() is called and the return value is initialized to lex. Then lookUp() is called with lex as its parameter.

* endedCorrectly()
  + endedCorrectly() first checks if any tokens have been initialized. If not an Error object is created (“no tokens initialized”). Then, if a token has been initialized, it checks if the source code given ended at the appropriate time. In this case, the last lexeme must be ‘end’. If the source code did not end correctly, an Error object is generated (“program ended unexpectedly”).
* printTokenTable(ArrayList<Token> tokens)
  + This function takes in an ArrayList of Tokens and prints out a table displaying all the token’s information: Its TokenType label and opcode, the lexeme value, and the line it occurs on.
* printErrorTable(ArrayList<Error> errors)
  + printErrorTable() takes in an ArrayList of Errors and prints out all Errors that occurred. The table contains the error message, the value that threw the error (if applicable), and the line that the error occurred on.

The main function creates the File object and calls readFile() to process it. It calls an initial getChar() to set the first nextChar and then loops lexer() over and over while currCharClass does not equal CharacterClass.EOF or an error has not occurred. After that it checks if the source code ended correctly by calling endedCorrectly(). Then if errors occurred printErrorTable() is called. Otherwise, it calls printTokenTable().

## Token.java

The Token class was primarily created to design a Token object. A Token object has three variables: a TokenType, a string, and an integer. The string stores the lexeme of the token, and the integer stores the line from the source code where the token appears. It has one method called printToken() that prints each token characteristic in a formatted way.

A TokenType is an enumerated class. It holds 24 enumerated constants that represent Julia keywords/reserved words and operations. Each enumerated constant has 3 properties: a String label, a String value, and an integer opcode. For example, if the lexeme is an assignment operator, or =, it would be assigned the ASSIGN\_OP constant. Its label would be “assignment operator”, its value would be “=”, and the opcode is 2000. The label and opcode are final, meaning they cannot be changed. The value is only constant for keywords/reserved words and operations. If the lexeme is a variable name or number, the value can be entered via the constructor.

The last part of the Token class is the CharacterClass enumeration. It has 4 constants: LETTER, DIGIT, UNKNOWN, and EOF. Each has one property, a String called description, that holds a description for each constant (for printing purposes). These constants help process characters from the source file as each character is given a CharacterClass. For example, if a character is a letter (A-Z or a-z) its character class is CharacterClass.LETTER. It also helps determine when the program should stop. If the CharacterClass is EOF, the character is an indicator that the source file has ended, and the program should not process anything past this point.

## Error.java

The Error class creates Error objects. Each Error object has three variables: a String msg, a String value, and an integer line. The msg variable holds the error message, the value String stores the value that caused the error if there is one applicable, and line stores the line of the source code where the error occurred.

There are three different constructors for an Error object. One where all values are initialized, one where msg and line are initiated, and one where only the msg is initiated.

It has only one method, printError(). printError prints all the values of an Error object in a formatted way.

# Screenshots (Successful runs and some errors)

## A computer screen capture Description automatically generated with medium confidenceTest1.jl Output

## A computer screen capture Description automatically generated with medium confidenceTest2.jl Output

## A screenshot of a computer Description automatically generated with medium confidenceTest3.jl

## A computer screen capture Description automatically generated with medium confidenceFile Not Found Error

## A screenshot of a computer Description automatically generated with medium confidenceLexeme is too long

## No tokens initialized error

## A screenshot of a computer Description automatically generated with medium confidenceProgram ended unexpectedly part 1

## A screenshot of a computer Description automatically generated with medium confidenceProgram ended unexpectedly part 2

## A screenshot of a computer Description automatically generated with medium confidenceUnexpected symbol

# References

Aei, et al. “In Java, How Can I Determine If a Char Array Contains a Particular Character?” *Stack Overflow*, Stack Exchange Inc, 2 Sept. 2013, https://stackoverflow.com/questions/18581531/in-java-how-can-i-determine-if-a-char-array-contains-a-particular-character.

Avagadro's Number, et al. “How Can I Store Newline Character in Char Variable?” *Stack Overflow*, Stack Exchange Inc., 17 Apr. 2017, https://stackoverflow.com/questions/43461412/how-can-i-store-newline-character-in-char-variable.

Baeldung. “Formatting Output with Printf() in Java.” *Baeldung*, Baeldung, 16 Sept. 2021, https://www.baeldung.com/java-printstream-printf.

“Compiler/Lexical Analyzer.” *Compiler/Lexical Analyzer - Rosetta Code*, GNU, 13 June 2021, https://rosettacode.org/wiki/Compiler/lexical\_analyzer.

duaprog137, et al. “How to Remove the Last Character from a String?” *Stack Overflow*, Stack Exchange Inc, 15 Sept. 2011, https://stackoverflow.com/questions/7438612/how-to-remove-the-last-character-from-a-string.

gowtham, Sai. “How to Get the Last Element of an Arraylist in Java.” *Reactgo*, Reactgo.com, 21 Oct. 2020, https://reactgo.com/java-get-last-element-of-arraylist/#:~:text=To%20get%20the%20last%20element%20of%20a%20ArrayList%20in%20Java,size()%2D1%20.

Gupta, Piyush. “Scanner and Nextchar() in Java.” *GeeksforGeeks*, GeeksforGeeks, 19 Oct. 2016, https://www.geeksforgeeks.org/gfact-51-java-scanner-nextchar/.

ina, et al. “Empty an Array in Java / Processing.” *Stack Overflow*, Stack Exchange Inc., 17 Nov. 2010, https://stackoverflow.com/questions/4208655/empty-an-array-in-java-processing.

“Java ArrayList to String ToString(ArrayList Characters).” *Java Arraylist to String Tostring(Arraylist Characters)*, java2s.Com, 2016, http://www.java2s.com/example/java-utility-method/arraylist-to-string/tostring-arraylist-character-characters-e8223.html.

“Java ArrayList.” *W3 Schools*, Refsnes Data, https://www.w3schools.com/java/java\_arraylist.asp.

“Java ArrayList.” *W3 Schools*, Refsnes Data, https://www.w3schools.com/java/java\_arraylist.asp.

“Java Convert Char to String - Javatpoint.” *Www.javatpoint.com*, https://www.javatpoint.com/java-char-to-string.

“Java Read Files.” *W3 Schools*, Refsnes Data, https://www.w3schools.com/java/java\_files\_read.asp.

Kasyap, Krishna. “How to Get First and Last Elements from ArrayList in Java?” *Tutorialspoint*, Tutorialspoint, 11 Oct. 2019, http://www.java2s.com/example/java-utility-method/arraylist-to-string/tostring-arraylist-character-characters-e8223.html.

Mathanlalsait. “Read a File Character by Character Using Java - Java File Handling.” *CandidJava Tutorial*, 14 Nov. 2019, https://www.candidjava.com/tutorial/program-to-read-a-file-character-by-character/.

Miglani, Gaurav. “Enum in Java.” *GeeksforGeeks*, GeeksforGeeks, 9 Sept. 2021, https://www.geeksforgeeks.org/enum-in-java/.

Muthu Ganapathy Nathan, et al. “How to Find Eof in a String in Java?” *Stack Overflow*, Stack Exchange Inc, 28 Sept. 2011, https://stackoverflow.com/questions/7590838/how-to-find-eof-in-a-string-in-java.

n0pe, et al. “Handling the Tab Character in Java.” *Stack Overflow*, Stack Exchange Inc., 24 May 2011, https://stackoverflow.com/questions/6112419/handling-the-tab-character-in-java.

Nitsdheerendra. “Conversion of Array to ArrayList in Java.” *GeeksforGeeks*, GeeksforGeeks, 4 Nov. 2019, https://www.geeksforgeeks.org/conversion-of-array-to-arraylist-in-java/.

Pan Long, et al. “Convert String to Arraylist in Java.” *Stack Overflow*, Stack Exchange Inc., 11 Mar. 2013, https://stackoverflow.com/questions/15331637/convert-string-to-arraylist-character-in-java/38395610.

phoenix, et al. “Converting Arraylist of Characters to a String?” *Stack Overflow*, Stack Exchange Inc, 12 June 2011, https://stackoverflow.com/questions/6324826/converting-arraylist-of-characters-to-a-string.

Piranha, Prickly. “Java Char Arraylist Code Example.” *Java Char Arraylist Code Example*, Grepper, 7 Nov. 2020, https://www.codegrepper.com/code-examples/java/java+char+arraylist.

Singh, Chaitanya. “Java String Startswith() Method with Example.” *Beginnersbook.com*, BeginnersBook, 3 Feb. 2019, https://beginnersbook.com/2013/12/java-string-startswith-method-example/.

user2062360, et al. “How Do I Compare a Character to Check If It Is Null?” *Stack Overflow*, Stack Exchange Inc, 13 Feb. 2013, https://stackoverflow.com/questions/14844793/how-do-i-compare-a-character-to-check-if-it-is-null.

user2276872, et al. “Tell If String Contains a-z Chars.” *Stack Overflow*, Stack Exchange Inc, 16 June 2014, https://stackoverflow.com/questions/24086968/tell-if-string-contains-a-z-chars.

Vayelin, et al. “Java Printf Formatting to Print Items in a Table or Columns.” *Stack Overflow*, Stack Exchange Inc, 1 Nov. 2015, https://stackoverflow.com/questions/33466526/java-printf-formatting-to-print-items-in-a-table-or-columns.