CMSC 214 Project 5

Derya O. Kurin

**Pseudocode:**

Define MatchGroupingSymbols class

Add main driver class which throws IOException

IF no args entered

Output: Source code file name not entered

Exit the program

ENDIF

DECLARE a file with command line args

IF file does not exist

Output: File doesn’t exist

Exit the program

ENDIF

DECLARE a Stack with characters called symbols

TRY Declaring a Scanner for file input

WHILE file has characters

Store the line in a String variable line

FOR each character in the line

Store each character in char variable: ch

IF ch is any of these symbols: ‘(’, ‘{‘, ‘[‘

Push that symbol character to the Stack

ELSE IF ch is any of these symbols: ‘)’, ‘}‘, ‘]‘

Call checkSymbolOrder to check if the closing symbols are in correct order

ENDIF

ENDFOR

ENDWHILE

ENDTRY

Write a Conditional operator >

IF symbols stack is empty,

Output: The Java source code file has correct symbol pairs.

ELSE

Output: The Java source code file does not have correct symbol pairs.

END Main Class

DEFINE METHOD: checkSymbolOrder

Parameters: stack: Stack of Characters, Character ch

Return type: void

IF stack.peek() is ‘(’ AND ch is ‘)’ OR

stack.peek() is ‘{’ AND ch is ‘}’ OR

stack.peek() is ‘[’ AND ch is ‘]’ OR

THEN stack.pop()

ELSE IF stack.peek() is not ‘(’ AND ch is ‘)’ OR

stack.peek() is not ‘{’ AND ch is ‘}’ OR

stack.peek() is not ‘[’ AND ch is ‘]’ OR

THEN Output: The Java source code does not have correct pairs.

Exit the program

ENDIF

END METHOD

END CLASS MatchGroupingSymbols

Screen shots:

InputFile.txt content: // Matching pairs

Graphical user interface, application, Teams

Description automatically generated

InputFile2.txt content: // Not matching pairs

Graphical user interface, text, application

Description automatically generated

Running the class on the Command Line with command line arguments:

Text

Description automatically generated

**Lessons Learned:**

With this project I learned that there are some scenarios where using stack make things much easier. For instance, if I did not create a Stack and store all the symbols that I would keep track of, then I would need to write much longer code to check if the symbols are immediately matching before getting any other symbols in between. Stack worked very well for this purpose since adding a symbol made that symbol at the peak and when I used pop method, I got rid of the symbol in the stack that has a matching pair.

At the end the logic went smooth when I finally checked if the stack is empty in other words all the symbols had a matching closing symbol.

One another thing I practiced with this example was to use the command line arguments for finding files. I initially had a little bit issue with finding the MatchingGroupingSymbol class with the java command but then after searching the error I got, I could solve my problem by running java command from one directory back with the package name prepended to the java class name. This solved my issue.

**Checklist:**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** |  | **Y/N** | **Comments** |
|  | Source java files | **Y** |  |
|  | Files: |  |  |
|  | LastNameFirstInitial\_Project05.zip | **Y** |  |
|  | LastNameFirstInitial\_Project05.doc | **Y** |  |
|  | Program compiles | **Y** |  |
|  | Program runs | **Y** |  |
|  | Checklist is completed and included in the Documentation | **Y** |  |
|  | Documentation file: |  |  |
|  | Comprehensive Test Plan | **Y** |  |
|  | Screenshots of running program | **Y** |  |
|  | ~~UML diagram~~ |  |  |
|  | Algorithm: Pseudo-code or Flowchart | **Y** |  |
|  | Lessons Learned | **Y** |  |
|  | Checklist | **Y** |  |