```
using System;
using System.Collections.Generic;
using System.Text;

namespace Complier
{
    /* Phillip Zamora
     * Lexical Analyzer
     * Used to get next token from String array
     */
    class LexicalAnalyzer
    {
        public enum symbol
        {
            classt, publict, statict, voidt, maint, stringt,
            extendst, returnt, intt, booleant, ift, elset,
            whilet, printt, lengtht, truet, falset, thist, newt,
            lParent, rParent, lBrackt, rBrackt, begint, endt,
            commat, semit, periodt, idt, numt, literalt, quotet,
            assignOpt, addOpt, mulOpt, relOpt, eoft, writet, unknownt,CCT
        }
        string[] lines;
        public List<symbol> Token= new List<symbol>();
        public List<string> lexemes= new List<string>();

        public string Lexeme;
        private char ch;
        public int value;
        public float valueR;
        public string Literal="";
        private ulong lineNumber = 0;
        private uint linePosition = 0;
        ArrayByEnum<string, symbol> resWords = new ArrayByEnum<string, symbol>();


//*************************************************************************
*******************************
        private List<string> reservedWords = new List<string> { "class",
"public","static","void","main",

"String","extends","return","int","boolean",
                                                                "if",
"else","while","System.out.println",

"length","true","false","this","new"};

//*************************************************************************
*******************************
        public LexicalAnalyzer(List<string> linesInput)
        {
```

```
    lines = linesInput.ToArray();
    resWords[symbol.begint] = "{";
    resWords[symbol.endt] = "}";
    resWords[symbol.maint] = "main";
    resWords[symbol.classt] = "class";
    resWords[symbol.publict] = "public";
    resWords[symbol.statict] = "static";
    resWords[symbol.voidt] = "void";
    resWords[symbol.stringt] = "String";
    resWords[symbol.extendst] = "extends";
    resWords[symbol.returnt] = "return";
    resWords[symbol.lengtht] = "length";
    resWords[symbol.thist] = "this";
    resWords[symbol.newt] = "new";
    resWords[symbol.ift] = "if";
    resWords[symbol.elset] = "else";
    resWords[symbol.intt] = "int";
    resWords[symbol.booleant] = "boolean";
    resWords[symbol.truet] = "true";
    resWords[symbol.falset] = "false";
    resWords[symbol.whilet] = "while";
    resWords[symbol.writet] = "System.out.println";

}
//Checks if string is in resWord list
public bool isReservedWord(string token)
{
    foreach(string resWord in reservedWords)
    {
        if (token == resWord)
        {
            return true;
        }
    }
    return false;
}
//Gets next token
public (string,string) GetNextToken()
{

    try
    {

        ch = lines[lineNumber][(int)linePosition];
    }
    catch
    {
        getNextCh();

        if (Token[Token.Count - 1] == symbol.eoft)
```

```
                {
                    return (Token[Token.Count - 1].ToString(), lexemes[lexemes.Count
- 1]);
                }
            }
            while (Char.IsWhiteSpace(ch)|| ch=='\0')
            {

                getNextCh();
                if (Token[Token.Count - 1] == symbol.eoft)
                {
                    return (Token[Token.Count - 1].ToString(), lexemes[lexemes.Count
- 1]);
                }
            }

            ProcessToken();


            return (Token[Token.Count - 1].ToString(), lexemes[lexemes.Count - 1]);

        }
        //Gets next char that isnt whitespace
        public void getNextCh()
        {
            linePosition++;
            try
            {
                // get next ch on linenumber and in line position
                ch = lines[lineNumber][(int)linePosition];
            }
            catch
            {
                // end of line so increment line and reset position
                lineNumber++;
                linePosition = 0;
                try
                {
                    ch = lines[lineNumber][(int)linePosition];
                }
                catch
                {

                    ch = ' ';
                    if (lines.Length < (int)lineNumber)
                    {
                        Lexeme = "";
                        lexemes.Add(Lexeme);
                        Token.Add(symbol.eoft);
                    }
```

```
            }

        }
    }
    //decides what token the lexeme is
    public void ProcessToken()
    {

        Lexeme = ch.ToString();
        getNextCh();
        try
        {
            if (Token[Token.Count - 1] == symbol.eoft)
            {
                return;
            }
        }
        catch { }

        if (Char.IsLetter(Lexeme[0]))
        {
            ProcessWordToken();
        }
        else if (Char.IsDigit(Lexeme[0]))
        {
            ProcessNumToken();
        }
        else if (Lexeme[0] == '/')
        {
            if (ch == '/' || ch == '*')
            {
                ProcessComment();
            }
            else
            {
                ProcessSingleToken();
            }
        }
        else if (Lexeme[0]== '='|| Lexeme[0] == '!' || Lexeme[0] == '>' ||
Lexeme[0] == '<')
        {
            if(ch == '=')
            {
                ProcessDoubleToken();
            }
            else
            {
                ProcessSingleToken();
            }
```

```
    }
    else if (Lexeme[0] == '|')
    {
        if (ch == '|')
        {
            ProcessORToken();
        }
        else
        {
            //ERROR
        }
    }
    else if (Lexeme[0] == '&')
    {
        if (ch == '&')
        {
            ProcessANDToken();
        }
        else
        {
            //ERROR
        }
    }
    else
    {
        ProcessSingleToken();
    }
}
//Check to see if AND token is valid
private void ProcessANDToken()
{
    Lexeme += ch;
    getNextCh();
    Token.Add(symbol.mulOpt);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
//Check to see if OR token is valid
private void ProcessORToken()
{
    Lexeme += ch;
    getNextCh();
    Token.Add(symbol.addOpt);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
//Process token if the lexme is a word
public void ProcessWordToken()
{
    readRestWord();
```

```
            if (Lexeme == "System" && ch=='.')
            {
                string temp = Lexeme;
                uint tempLinPos = linePosition;
                Lexeme += ch;
                getNextCh();
                readRestWord();
                if (Lexeme == "System.out" && ch == '.')
                {
                    Lexeme += ch;
                    getNextCh();
                    readRestWord();
                    if (Lexeme != "System.out.println")
                    {
                        Lexeme = temp;
                        linePosition = tempLinPos-1;
                        getNextCh();
                    }
                }

            }

            foreach (symbol sym in Enum.GetValues(typeof(symbol)))
            {
                if (resWords[sym] == Lexeme)
                {
                    Token.Add(sym);
                    lexemes.Add(Lexeme);
                    Lexeme = "";
                    return;
                }
            }
            if (Lexeme.Length > 31)
            {
                Token.Add(symbol.unknownt);
                Lexeme ="ERROR Line "+lineNumber + ": "+Lexeme+ "  ID Token has size
limit of 31";
                lexemes.Add(Lexeme);
            }
            else
            {
                Token.Add(symbol.idt);
                lexemes.Add(Lexeme);
            }

            Lexeme = "";
            return;
```

```
}
//Get the literal
private void getLiteral()
{
     var temp = linePosition;
    while (ch != '"')
    {
        this.Literal += ch;
        Lexeme += ch;
        getNextCh();
        if (temp > linePosition)
        {
            getNextCh();
            return;
        }
    }
    Lexeme += ch;
    getNextCh();
}
//Fills the rest of the lexeme
public void readRestWord()
{
    while(ch=='_'|| Char.IsLetterOrDigit(ch))
    {
        var temp = linePosition;
        Lexeme += ch;
        getNextCh();
        if (temp > linePosition)
        {
            return;
        }
    }

}
//Process if lexmem is number
public void ProcessNumToken()
{
    readRestNum();
    Token.Add(symbol.numt);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
//fill the rest of the lexme with numbers
private void readRestNum()
{
    int count = 0;
    while (ch == '.' || Char.IsDigit(ch))
    {
        if (ch == '.')
        {
```

```
            count++;
        }

        if (count > 1)
        {
            return;
        }
        var temp = linePosition;
        Lexeme += ch;
        getNextCh();
        if (temp > linePosition)
        {
            return;
        }


    }
}
//processes if it is a comment
public void ProcessComment()
{
    if (ch == '/')
    {
        lineNumber++;
        linePosition = 0;
        try
        {
            ch = lines[lineNumber][(int)linePosition];
        }
        catch
        {
            getNextCh();
        }

    }
    else
    {
        findEndOfComment();
    }
    Lexeme = "";

    ProcessToken();

}
// continues to look for end of comment
private void findEndOfComment()
{
    bool end = false;
    while (!end)
    {
```

```
        if (Token[Token.Count-1] == symbol.eoft)
        {
            return;
        }
        else if (ch == '*')
        {
            getNextCh();
            if (ch == '/')
            {

                end = true;
                getNextCh();
            }
        }
        else
        {
            getNextCh();
        }
    }
}
// process if the token has 2 part
public void ProcessDoubleToken()
{
    Lexeme += ch;
    getNextCh();
    Token.Add(symbol.relOpt);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
//process if lexme is single char
public void ProcessSingleToken()
{
    if (Lexeme[0] == '=')
    {
        Token.Add(symbol.assignOpt);
        lexemes.Add(Lexeme);
        Lexeme = "";
    }
    else if (Lexeme[0] == '+' || Lexeme[0] == '-')
    {
        Token.Add(symbol.addOpt);
        lexemes.Add(Lexeme);
        Lexeme = "";
    }
    else if (Lexeme[0] == '*' || Lexeme[0] == '/')
    {
        Token.Add(symbol.mulOpt);
        lexemes.Add(Lexeme);
        Lexeme = "";
    }
```

```
else if (Lexeme[0] == '"')
{
    getLiteral();
    if (Literal[Literal.Length - 1] == Lexeme[Lexeme.Length - 1])
    {
        Token.Add(symbol.unknownt);
    }
    else
    {
        Token.Add(symbol.literalt);
    }
    lexemes.Add(Lexeme);
    Lexeme = "";
    Literal = "";
}
else if(Lexeme[0] == '(')
{
    Token.Add(symbol.lParent);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
else if (Lexeme[0] == ')')
{
    Token.Add(symbol.rParent);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
else if (Lexeme[0] == '{')
{
    Token.Add(symbol.begint);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
else if (Lexeme[0] == '}')
{
    Token.Add(symbol.endt);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
else if (Lexeme[0] == ',')
{
    Token.Add(symbol.commat);
    lexemes.Add(Lexeme);
    Lexeme = "";
}
else if (Lexeme[0] == ';')
{
    Token.Add(symbol.semit);
    lexemes.Add(Lexeme);
    Lexeme = "";
```

```
            }
            else if (Lexeme[0] == '.')
            {
                Token.Add(symbol.periodt);
                lexemes.Add(Lexeme);
                Lexeme = "";
            }
            else if (Lexeme[0] == '[')
            {
                Token.Add(symbol.lBrackt);
                lexemes.Add(Lexeme);
                Lexeme = "";
            }
            else if (Lexeme[0] == ']')
            {
                Token.Add(symbol.rBrackt);
                lexemes.Add(Lexeme);
                Lexeme = "";
            }
            else if (Lexeme[0] == '>')
            {
                Token.Add(symbol.relOpt);
                lexemes.Add(Lexeme);
                Lexeme = "";
            }
            else if (Lexeme[0] == '<')
            {
                Token.Add(symbol.relOpt);
                lexemes.Add(Lexeme);
                Lexeme = "";
            }
            else
            {
                Token.Add(symbol.unknownt);
                lexemes.Add(Lexeme);
                Lexeme = "";
            }

        }

    }
}
```