

```
package analyzer;
import java.io.*;
/**
 * Class that Tokenizes the file
 * @author Sergio Alcocer
 *
 */
public class Lexer implements TokenConstants {
    private FileLoader stream;

    /**
     * @param filename String with the path of the file to tokenize
     * @throws IOException in case something wrong happens
     */
    public Lexer(String filename) throws IOException {
        this.stream = new FileLoader(new File(filename));
    }

    /**
     * Gets the next token
     * @return next Token in the file
     */
    public Token getNextToken() {
        Token nextToken = tokenize();
        while(nextToken == null) nextToken = tokenize();
        return nextToken;
    }

    /**
     * Closes the file
     */
    public void close() {
        this.stream.close();
    }

    /**
     * Using an automaton, gets characters until it can, then backtracks until the longest Token
     * @return next Token (including null Tokens)
     */
    private Token tokenize() {
```

```
int finalState = -1;
StringBuffer lexeme = new StringBuffer();
StringBuffer tainting = new StringBuffer();
char newChar = stream.getNextChar();
int state = transition(0,newChar);
int row = stream.getRow();
int column = stream.getColumn();
while(state!=-1 && newChar != '\0') {
    tainting.append(newChar);
    if(isFinal(state)) {
        finalState = state;
        lexeme.append(tainting);
        tainting.delete(0,tainting.length());
    }
    newChar = stream.getNextChar();
    state = transition(state,newChar);
}
if(finalState != -1) {
    stream.retract(1+tainting.length());
    return getToken(finalState,lexeme.toString(),row,column);
} else if(newChar != '\0') {
    stream.retract(tainting.length());
    throw new LexicalError(newChar,row,column);
} else {
    stream.retract(1);
    return new Token(Token.EOF,"",row,column);
}
}

/**
 * Implements Token's Automaton
 * @param state current state in the automaton
 * @param symbol character read
 * @return next state
 */
protected int transition(int state, char symbol) {
    switch(state) {
        case 0:
            if (symbol == '\n' || symbol == '\t' || symbol == '\r' || symbol == ' ') return 1;
            else if (symbol == '"') return 2;
```

```
        else if (symbol == '\\') return 5;
        else if (symbol == '/') return 8;
        else if (symbol == '{') return 14;
        else if (symbol == '}') return 15;
        else if (symbol == '(') return 16;
        else if (symbol == ')') return 17;
        else if (symbol == '_' || (symbol >= 'a' && symbol <= 'z') || (symbol >= 'A' && symbol <= 'Z') || (symbol >=
'0' && symbol <= '9')) return 18;
        else if (symbol == ';') return 19;
        else return 1;
    case 2:
        if (symbol == '\\') return 3;
        else if (symbol == '"') return 4;
        else return 2;

    case 3:
        return 2;

    case 5:
        if (symbol == '\\') return 6;
        else return 7;

    case 6:
        return 7;

    case 8:
        if (symbol == '*') return 9;
        else if (symbol == '/') return 12;
        else return -1;

    case 9:
        if (symbol == '*') return 10;
        else return 9;

    case 10:
        if (symbol == '*') return 10;
        else if (symbol == '/') return 11;
        else return 9;

    case 12:
```

```
    if (symbol == '\n') return 13;
    else return 12;
```

```
    case 18:
```

```
        if (symbol == '_' || (symbol >= 'a' && symbol <= 'z') || (symbol >= 'A' && symbol <= 'Z') || (symbol >= '0' &&
symbol <= '9') || symbol == '.') return 18;
```

```
        else return -1;
```

```
    default:
```

```
        return -1;
```

```
    }
```

```
}
```

```
/**
```

```
 * Check if a state is final or not
```

```
 * @param state state
```

```
 * @return true, if the state is final
```

```
 */
```

```
protected boolean isFinal(int state) {
```

```
    if(state <=0 || state > 19) return false;
```

```
    switch(state) {
```

```
        case 1:
```

```
        case 4:
```

```
        case 7:
```

```
        case 8:
```

```
        case 11:
```

```
        case 13:
```

```
        case 14:
```

```
        case 15:
```

```
        case 16:
```

```
        case 17:
```

```
        case 18:
```

```
        case 19:
```

```
            return true;
```

```
    default:
```

```
        return false;
```

```
    }
```

```
}
```

```
/**
```

```
 * Generates the lexical component associated to the final state and
```

```
* to the lexeme found. Returns null if the action has to be skip
*
* @param state Final state
* @param lexeme recognized lexeme
* @param row
* @param column
* @return Lexical component associated
*/
protected Token getToken(int state, String lexeme, int row, int column) {
    switch(state) {
        case 1:
        case 4:
        case 7:
        case 8:
        case 11:
        case 13:    return null;
        case 14:    return new Token(OPEN_CURLY           , lexeme, row, column);
        case 15:    return new Token(CLOSE_CURLY          , lexeme, row, column);
        case 16:    return new Token(OPEN_BRACK           , lexeme, row, column);
        case 17:    return new Token(CLOSE_BRACK          , lexeme, row, column);
        case 18:    if (lexeme.equals("import")) return new Token(IMPORT, lexeme, row, column);
                    else if (lexeme.equals("package")) return new Token(PACKAGE, lexeme, row, column);
                    else if (lexeme.equals("public")) return new Token(PUBLIC, lexeme, row, column);
                    else if (lexeme.equals("private")) return new Token(PRIVATE, lexeme, row, column);
                    else if (lexeme.equals("protected")) return new Token(PROTECTED, lexeme, row, column);
                    else if (lexeme.equals("abstract")) return new Token(ABSTRACT, lexeme, row, column);
                    else if (lexeme.equals("static")) return new Token(STATIC, lexeme, row, column);
                    else if (lexeme.equals("final")) return new Token(FINAL, lexeme, row, column);
                    else if (lexeme.equals("class")) return new Token(CLASS, lexeme, row, column);
                    else return new Token(VALUE, lexeme, row, column);

        case 19:    return new Token(SC                     , lexeme, row, column);
        default:    return null;
    }
}

/**
 * Testing main
 * @param args [0] filename
 */
```

```
public static void main(String[] args) {
    int _ = 0;
    System.out.println(_ + "123");
    if(args.length == 0){ System.out.println("no file"); return;}

    try {
        Lexer lexer = new Lexer(args[0]);
        Token tk;
        do {
            tk = lexer.getNextToken();
            System.out.println(tk.toString());
        } while(tk.getKind() != Token.EOF);
    } catch(Exception ex) {
        ex.printStackTrace();
    }
}
```