Contents

1 Basic Test Results

```
printing files in /tmp/bodek.eeZu0e/oop/ex6/muaz.abdeen/presubmission/testdir/30722
    README
    submission
    oop
    META-INF
Logins: muaz.abdeen

Logins: mmda1282



compiling with
    javac -Xlint:rawtypes -Xlint:empty -Xlint:divzero -Xlint:deprecation -cp .:/cs/course/current/oops/lib/junit4.jar *.java


tests output :
        Testing 501
Testing 502
Testing 503
Testing 504
Testing 505
Perfect!
```

2 README

```
muaz.abdeen
mmda1282
ex6
============================
=      File description     =
============================
## oop.ex6 Package ##

= main subpackage =
    - Sjavac - Main class: runs the verifier on the source file.

= symbol subpackage =
    - Variable - A class represents a variable.
    - Block - A class represents a Block of code (= method or if\while).

= verifier subpackage =
    - FileVerifier - A class combine all partial verifiers.
    - SemanticsVerifier -  verifier class for static semantics of the code:
                           checks valid assignments and referring for variables.
    - SyntaxVerifier - a class verities syntax correctness.
    - SymbolTable - Class of symbol tables of the code.

- FileReader - Reads whole file content to an array.
- Parser - Parses each line into tokens.
- SJavaRegex - class contains all of regex patterns of the code.
- VerifierExceptions - Exceptions class for the code verifier.


============================
=          Design          =
============================

(1) Verifier Exceptions
    We create single class that extends the Exception class, VerifierExceptions.
    We put then in the source package, i.e. oop.ex6 package, because different
    classes from different packages in the source package used the exceptions.

(2) Verifier
    We did not notice an obvious hierarchy between the verifier classes, although
    they doing a consequent stages of the verifying process, they did not sharing
    code, so we put them together in the same package but without inheritance.

(3) Symbol
    We look to the code as if it constructed from elementary components: variables,
    and complex ones, blocks. Like verifier, we did not notice an obvious hierarchy
    between them, and no code sharing, so we just put them in the same package.

(4) Regex
    We put all of regex patterns of the code in this class, trying to make it
    similar to a factory class.


============================
=  Implementation details  =
============================

(1) Handling exceptions:
    I handled the IO exceptions in the top file verifier class, in which
```

```
    we tried to reach the source file.
    The Verifying exceptions were thrown in different stages and classes,
    and delegated (rethrown) to the top verifier class, where were handled.

(2) Regex:
    We put all of regex patterns of the code in one class, these regex patterns
    varies: there is a whole line regex, and variable regex, method regex, and
    if\while regex.

(3) Parser:
    parses every line to the suitable tokens, and prepares these tokens to be
    used in other classes, like building symbol tables, and verifying semantics.

(4) Verifying:
    We decided to loop over the code three times: in the first loop we check syntax legality,
    using the suitable regex patterns for legal lines:
        /* Legal lines are:
           (1) comment: //
           (2) varDec: type keyword or final (many separated by commas)
           (3) varAssign: name (dont allow name to be keyword)
           (4) varRef: varDec or varAssign
           (5) methodDef: void keyword
           (6) methodCall: name()
           (7) returnLine: return
           (8) blocks: if \ while keyword
           (9) closingBracket: }
         */
    In the second loop we build the global symbol tables: the method table, which contains
    all method signatures, and the global variable table, which contains all global variables,
    then we put the global variable table as the root node of a linked list of scope variable
    tables.
    In the third loop: we check every method in the methods table, and building a new local
    symbol table when entering the scope and append it to the linked list of scope variable
    tables, and delete it when exiting the scope.
    These symbol tables were used to verify semantics, i.e. type compatibility, and values
    legality (legal assignments).


============================
=    Answers to questions   =
============================
Q - How you handles-Java code errors in this exercise, and why you chose to do so?
A - We handled the IO exceptions in the top file verifier class, in which
    we tried to reach the source file.
    The Verifying exceptions were thrown in different stages and classes,
    and delegated (rethrown) to the top verifier class, where were handled.

Q - How would you modify your code to add new types of variables (e.g.,float)?
A - define their regex patterns in the Regex class, this will automatically make this
    type legal, and parser will parses the line contains it, and will be added to
    symbol tables.

Q - Describe which modifications/extensions you would have to make in your code in
    order to support: Classes, Different methods' types.
A - (1) Classes: since classes are kind of block, so we first define the legal regex
        pattern of class signature, and using the block class to store its details,
        like (access modifiers, name, etc.), and in the verifier class we will, define
        a new method to start verifying the class, similar to method verifying, loop over
        lines and check legality of each line according to sjava class rules.
        if multiple classes can be define in the same file, we will add classes table to
        symbol table class.

    (2) Different methods' types: change the regex pattern of method declaration, making
        types other than void legal, and add new attribute to method block, which is type,
        and check return value and type compatibility similar to variable assignment.

Q - Describe two of the main regular expressions you used in your code.
A - (1) VARIABLE_DECLARATION
```

```
        "(" + FINAL + ")(" + TYPE + ")(?:\\s*" + OPTIONAL_ASSIGNMENT + "\\s*,)*\\s*" +
        OPTIONAL_ASSIGNMENT + "\\s*;\\s*$"
        a. FINAL = "\\s*(?:final\\s+)?"  =  matches zero or more whitespaces, then the optional group:
                                            the word final literally and one or more whitespaces.
        b. TYPE = "(?:int|double|boolean|char|String)\\s+" = exactly one of these words then one or
                                                             more whitespaces.
        c. OPTIONAL_ASSIGNMENT = "(?:(" + VARIABLE_NAME + ")(?:" + ASSIGNING_OPERATOR +
                                 "(" + VARIABLE_VALUE + "))?)"
                               = matches the variable name, then the assigning operator, then the
                                 optional group variable name.
        d. "\\s*;\\s*$" = zero or more whitespaces, then the ; symbol, then zero or more whitespaces,
                          at end of the line.

    (2) METHOD_DEFINITION
        "^\\s*void\\s*(?<name>" + METHOD_NAME + ")\\s*" + METHOD_PARAMETER_LIST + "\\s*\\{\\s*$"
        a. "^\\s*void\\s*" = from the beginning of the line, zero or more whitespaces, then the
                             word void literally, then zero or more whitespaces.
        b. "(?<name>" + METHOD_NAME + ")\\s*" = group named: name, matches a predefined regex
                                                called method name, then zero or more whitespaces.
        c. "\\s*\\{\\s*$"  = zero or more whitespaces, then the { symbol, then zero or more whitespaces,
                             at end of the line.
```

3 oop/ex6/FileReader.java

```java
package oop.ex6;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

/**
 * Reads whole file content to an array
 */
public class FileReader {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    /** Array contains all input file lines */
    private static String[] fileContent;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Construct A sjava file reader.
     * Reads the complete file (all lines) at once into a string array.
     * @param inputFile file to read
     * @throws IOException Error in handling source file
     */
    public FileReader(String inputFile) throws IOException {
        try {
            File input = new File(inputFile);
            FileInputStream fileInputStream = new FileInputStream(input);
            byte[] inputData = new byte[(int) input.length()];
            fileInputStream.read(inputData);
            fileInputStream.close();
            fileContent = new String(inputData).split("\n");

        } catch (Exception e) {
            throw new IOException();
        }
    }

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * gets the file content array
     * @return array of strings contains all file's lines
     */
    public String[] getFileContent() {
        return fileContent;
    }

}
```

4 oop/ex6/Parser.java

```java
package oop.ex6;

import oop.ex6.symbol.Block;
import oop.ex6.symbol.Variable;

import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Parses each line into tokens
 */
public class Parser {

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    /** current parsed code line */
    String currentLine;
    /** current parsed token of the line */
    String currentToken;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Constructs a new line parser
     * @param line valid code line (already syntactically checked)
     */
    public Parser(String line) {
        this.currentLine = line;
    }

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * checks if line has more tokens.
     * @return true it there is more tokens, else false.
     */
    private boolean hasMoreTokens() {
        return currentLine.length() > 0;
    }

    /**
     * advance to the next token.
     */
    private void advance() {
        currentLine = currentLine.substring(currentToken.length()).trim();
    }

    /**
     * parses a variable line into list of details lists, details list
     * for every variable in this line:
     *     {{name, constant, type, value}, ...}
     * @return ArrayList of variables details.
```

```
60          * @throws VerifierExceptions invalid line
61          */
62         public ArrayList<String[]> parseVarLine() throws VerifierExceptions {
63             ArrayList<String[]> lineVars = new ArrayList<>();
64             // I. parse final and type
65             String constant = null, type = null;
66             Pattern varPattern =
67                     Pattern.compile("(?<final>^" + SJavaRegex.FINAL + ")(?<type>" + SJavaRegex.TYPE + ")");
68             Matcher varMatcher = varPattern.matcher(currentLine);
69             if (varMatcher.find()) {
70                 constant = varMatcher.group("final").trim();
71                 constant = constant.equals("") ? null : constant;
72                 type = varMatcher.group("type").trim();
73                 currentToken = varMatcher.group();
74                 advance();
75             }
76             // II. parse name and value
77             while (hasMoreTokens()) {
78                 String[] NameVal = getNameAndValue();
79                 String name = NameVal[0], value = NameVal[1];;
80                 lineVars.add(new String[]{name, constant, type, value});
81                 advance();
82             }
83             return lineVars;
84         }
85
86         /**
87          * get 2 slot string array contains the variable's name and value if exists.
88          * @return String array of name nad value
89          */
90         private String[] getNameAndValue() throws VerifierExceptions {
91             // I. declaration only
92             Pattern varDecPattern = Pattern.compile("^(?:" + SJavaRegex.VARIABLE_NAME + ")\\s*[;,]");
93             Matcher varDecMatcher = varDecPattern.matcher(currentLine);
94             if (varDecMatcher.find()) {
95                 currentToken = currentLine.substring(varDecMatcher.start(), varDecMatcher.end());
96                 String name = currentToken.replaceAll("[;,]|\\s*", "");
97                 if (SJavaRegex.KEYWORDS.matcher(name).matches())
98                     throw new VerifierExceptions(VerifierExceptions.ILLEGAL_NAME);
99                 return new String[]{name, null};
100            }
101            // II. assignment
102            Pattern varAssignPattern = Pattern.compile("(?:\\s*" + SJavaRegex.VARIABLE_NAME + ")\\s*=\\s*" +
103                                                       "(?:" + SJavaRegex.VARIABLE_VALUE + ")\\s*[,;]");
104            Matcher varAssignMatcher = varAssignPattern.matcher(currentLine);
105            if (varAssignMatcher.find()) {
106                currentToken = currentLine.substring(varAssignMatcher.start(), varAssignMatcher.end());
107                int idx = currentToken.indexOf('=');
108                String name = currentToken.substring(0, idx).trim();
109                String value = currentToken.substring(idx+1).replaceFirst("[;,]", "").trim();
110                if (SJavaRegex.KEYWORDS.matcher(name).matches())
111                    throw new VerifierExceptions(VerifierExceptions.ILLEGAL_NAME);
112                return new String[]{name, value};
113            }
114            return new String[2];
115        }
116
117        /**
118         * return method line details: name and (arguments)*
119         * syntax is already checked: void name(arg*) {
120         * @return method object contains method declaration details
121         * @throws VerifierExceptions illegal method
122         */
123        public Block parseMethodSignature() throws VerifierExceptions {
124            String methodName = null;
125            ArrayList<Variable> arguments = new ArrayList<>();
126            Pattern namePattern = Pattern.compile("^\\s*void\\s*(?<name>" + SJavaRegex.METHOD_NAME +")\\s*\\(");
127            Matcher nameMatcher = namePattern.matcher(currentLine);
```

```
128             if (nameMatcher.find())
129                 methodName = nameMatcher.group("name").trim();
130             assert methodName != null;
131             if (SJavaRegex.KEYWORDS.matcher(methodName).matches())
132                 throw new VerifierExceptions(VerifierExceptions.ILLEGAL_NAME);
133 
134             Pattern parameter = Pattern.compile("\\s*(?<final>final\\s+)?" +
135                                                 "(?<type>int|double|boolean|char|String)\\s+" +
136                                                 "(?<name>(?:_\\w|[a-zA-Z])\\w*)");
137             Matcher paramMatcher = parameter.matcher(currentLine);
138             while (paramMatcher.find()) {
139                 boolean constant = paramMatcher.group("final") != null;
140                 String type = paramMatcher.group("type").trim();
141                 String name = paramMatcher.group("name").trim();
142                 if (SJavaRegex.KEYWORDS.matcher(name).matches())
143                     throw new VerifierExceptions(VerifierExceptions.ILLEGAL_NAME);
144                 // check parameters with same name
145                 for (Variable arg : arguments) {
146                     if (arg.name().equals(name))
147                         throw new VerifierExceptions(String.format(VerifierExceptions.SAME_PARAMETERS,
148                                                                    methodName));
149                 }
150                 Variable arg = new Variable(name, type, null, constant);
151                 arg.markAsArg();
152                 arguments.add(arg);
153             }
154             return new Block(methodName, arguments);
155         }
156 
157         /**
158          * return method call line details
159          *     methodName((parameters,)* parameter?)
160          *     parameter = name | value
161          * @return ArrayList contains the line details
162          */
163         public ArrayList<String> parseCallLine() {
164             ArrayList<String> lineDetails = new ArrayList<>();
165             Pattern namePattern = Pattern.compile("^\\s*(?<name>" + SJavaRegex.METHOD_NAME +")\\s*\\(");
166             Matcher nameMatcher = namePattern.matcher(currentLine);
167             if (nameMatcher.find())
168                 lineDetails.add(nameMatcher.group("name"));
169             currentToken = currentLine.substring(nameMatcher.start(), nameMatcher.end());
170             advance();
171             Pattern methodArg = Pattern.compile("(?<arg>" + SJavaRegex.ARGUMENT + ")[\\s*|,|\\)]");
172             Matcher argMatcher = methodArg.matcher(currentLine);
173             while (argMatcher.find()) {
174                 lineDetails.add(argMatcher.group("arg"));
175             }
176             return lineDetails;
177         }
178 
179         /**
180          * Parses the if\while line into a list of its conditions
181          * @return ArrayList contains the line conditions
182          */
183         public ArrayList<String> parseIfWhileLine() {
184             ArrayList<String> lineDetails = new ArrayList<>();
185             int idx = currentLine.indexOf('(');
186             currentLine = currentLine.substring(idx);
187             Pattern methodArg = Pattern.compile("(?<condition>" + SJavaRegex.CONDITION + ")");
188             Matcher argMatcher = methodArg.matcher(currentLine);
189             while (argMatcher.find()) {
190                 lineDetails.add(argMatcher.group("condition"));
191             }
192             return lineDetails;
193         }
194 
195     }
```

5 oop/ex6/SJavaRegex.java

```java
package oop.ex6;

import java.util.regex.Pattern;

/**
 * class contains all of regex patterns of the code.
 */
public class SJavaRegex {
    // The metacharacters are: <([{\^-=$!|]})?*+.>

    // ########################
    // ##  SJAVA LINE REGEX  ##
    // ########################

    private final static String CODE_KEYWORDS = "int|double|boolean|char|String|void|final|" +
                                                "if|while|true|false|return";

    private final static String EMPTY_LINE_REGEX = "^\\s*$";

    private final static String COMMENT_LINE_REGEX = "^//.*\\s*$";
    private final static String ILLEGAL_INLINE_COMMENT = "^.+//.*$";
    private final static String ILLEGAL_MULTILINE_COMMENT = "^.*(?:(?:/\\*|\\s*\\*|.*\\*/).*$)";
    private final static String ILLEGAL_COMMENT = ILLEGAL_INLINE_COMMENT + "|" +
                                                  ILLEGAL_MULTILINE_COMMENT;

    private final static String LINE_SUFFIX = "\\s*$";
    private final static String CLOSE_SUFFIX_REGEX = "^\\s*\\}\\s*$";

    private final static String ARRAYS_BRACKETS = "\\[\\]";
    private final static String ILLEGAL_OPERATORS = "\\+\\-\\*/%!";
    private final static String OPERATOR_OR_ARRAYS = "[" + ILLEGAL_OPERATORS + ARRAYS_BRACKETS + "]";

    // ######################
    // ##  VARIABLE REGEX  ##
    // ######################

    public final static String FINAL = "\\s*(?:final\\s+)?";

    public final static String TYPE = "(?:int|double|boolean|char|String)\\s+";

    public final static String VARIABLE_NAME = "(?:_\\w|[a-zA-Z])\\w*";

    private final static String ASSIGNING_OPERATOR = "\\s*\\=\\s*";

    public final static String INT_VALUE = "-?\\d++";
    public final static String DOUBLE_VALUE = "-?\\d+(?:\\.\\d++)?";
    public final static String BOOL_VALUE = "true|false|(?:" + DOUBLE_VALUE + ")";
    public final static String CHAR_VALUE = "'.'";
    public final static String STRING_VALUE = "\".*\"";

    public final static String VARIABLE_VALUE =
            INT_VALUE + "|" + DOUBLE_VALUE + "|(?:" + BOOL_VALUE + ")|" + CHAR_VALUE + "|" + STRING_VALUE +
            "|(?:" + VARIABLE_NAME + ")";

    public final static String OPTIONAL_ASSIGNMENT = "(?:(" + VARIABLE_NAME + ")(?:" + ASSIGNING_OPERATOR +
                                                     "(" + VARIABLE_VALUE + "))?)";
    private final static String VAR_DEC_REGEX =
            "(" + FINAL + ")(" + TYPE + ")(?:\\s*" + OPTIONAL_ASSIGNMENT + "\\s*,)*\\s*"
            + OPTIONAL_ASSIGNMENT + "\\s*;\\s*$";
```

```
60
61      private final static String VAR_ASSIGN_REGEX =
62              "^\\s*(?:" + VARIABLE_NAME + ")" + ASSIGNING_OPERATOR + "(?:" + VARIABLE_VALUE + ")\\s*;\\s*$";
63
64      // ####################
65      // ##  METHOD REGEX  ##
66      // ####################
67
68      public final static String METHOD_NAME = "[a-zA-Z]\\w*";
69      public final static String METHOD_PARAMETER = FINAL + TYPE + VARIABLE_NAME;
70      private final static String METHOD_PARAMETER_LIST = "\\((?:" + METHOD_PARAMETER + "\\s*,)*\\s*(?:" +
71                                                          METHOD_PARAMETER + ")?\\s*\\)";
72      private final static String METHOD_DEF_REGEX = "^\\s*void\\s*(?<name>" + METHOD_NAME + ")\\s*" +
73                                                     METHOD_PARAMETER_LIST + "\\s*\\{\\s*$";
74
75      public final static String ARGUMENT = VARIABLE_VALUE + "|(?:" + VARIABLE_NAME + ")";
76      private final static String ARGUMENTS = "\\(\\s*(?:(?:\\s*(?:" + ARGUMENT + ")\\s*,)*\\s*" +
77                                              "\\s*(?:" + ARGUMENT + "))?\\s*\\)";
78
79      private final static String METHOD_CALL_REGEX = "^\\s*" + METHOD_NAME + "\\s*" +
80                                                      ARGUMENTS + "\\s*;\\s*$";
81      private final static String RETURN = "^\\s*return\\s*;\\s*$";
82
83      // ######################
84      // ##  IF/WHILE REGEX  ##
85      // ######################
86
87      private final static String BLOCK = "(?:if|while)";
88      public final static String CONDITION = "(?:" + BOOL_VALUE + "|(" + VARIABLE_NAME + "))";
89      private final static String AND_OR = "(?:\\|\\||&&)";
90      private final static String MULTIPLE_CONDITION =
91              "(?:(" + CONDITION + "\\s*" + AND_OR + "\\s*)*\\s*" + CONDITION + ")";
92      private final static String IF_WHILE_REGEX = "^\\s*" + BLOCK + "\\s*\\(\\s*" + MULTIPLE_CONDITION +
93                                                   "\\s*\\)" + "\\s*\\{\\s*$";
94
95
96      // ############################
97      // ##  LEGAL LINES PATTERNS  ##
98      // ############################
99      /* Legal lines are:
100        (1) comment: //
101        (2) varDec: type keyword or final (many separated by commas)
102        (3) varAssign: name (dont allow name to be keyword)
103        (4) varRef: varDec or varAssign
104        (5) methodDef: void keyword
105        (6) methodCall: name()
106        (7) returnLine: return
107        (8) blocks: if \ while keyword
108        (9) closingBracket: }
109     */
110
111     /** KEYWORDS */
112     public static final Pattern KEYWORDS = Pattern.compile(SJavaRegex.CODE_KEYWORDS);
113
114     /** LEGAL LINES */
115     public static final Pattern EMPTY_LINE = Pattern.compile(SJavaRegex.EMPTY_LINE_REGEX);
116     public static final Pattern COMMENT_LINE = Pattern.compile(SJavaRegex.COMMENT_LINE_REGEX);
117     public static final Pattern VARIABLE_DECLARATION = Pattern.compile(SJavaRegex.VAR_DEC_REGEX);
118     public static final Pattern VARIABLE_ASSIGNMENT = Pattern.compile(SJavaRegex.VAR_ASSIGN_REGEX);
119     public static final Pattern METHOD_DEFINITION = Pattern.compile(SJavaRegex.METHOD_DEF_REGEX);
120     public static final Pattern METHOD_CALL = Pattern.compile(SJavaRegex.METHOD_CALL_REGEX);
121     public static final Pattern RETURN_STATEMENT = Pattern.compile(SJavaRegex.RETURN);
122     public static final Pattern IF_WHILE_BLOCK = Pattern.compile(SJavaRegex.IF_WHILE_REGEX);
123     public static final Pattern CLOSE_SUFFIX = Pattern.compile(SJavaRegex.CLOSE_SUFFIX_REGEX);
124
125     /** Legal patterns array */
126     public static final Pattern[] LEGAL_PATTERN = {EMPTY_LINE, COMMENT_LINE,
127                                                    VARIABLE_DECLARATION, VARIABLE_ASSIGNMENT,
```

```
128                                                 METHOD_DEFINITION, METHOD_CALL,
129                                                 RETURN_STATEMENT, CLOSE_SUFFIX,
130                                                 IF_WHILE_BLOCK};
131
132
133   }
```

6 oop/ex6/VerifierExceptions.java

```java
package oop.ex6;

/**
 * Exceptions class for the code verifier
 */
public class VerifierExceptions extends Exception{
    private static final long serialVersionUID = 1L;

    // ############################ //
    // #### SYNTAX EXCEPTIONS #### //
    // ############################ /

    public static final String BAD_SYNTAX_MSG = "bad syntax: %s %n";
    public static final String BAD_BRACKETS = "bad syntax: " +
                                              "illegal closing bracket, no corresponding opening %n";
    public static final String BAD_BLOCKS_MSG = "ERROR: \n\tbad syntax, check blocks brackets. \n";
    public static final String GLOBAL_IF_WHILE = "if\\while block not allowed in the global scope %n";

    public static final String ILLEGAL_NAME = "Sjava keyword is not an illegal name %n";


    // ############################### //
    // #### VARIABLE EXCEPTIONS #### //
    // ############################### //

    public static final String BAD_REFERENCE = "referred to undeclared variable: %s %n";
    public static final String INCOMPATIBLE_TYPE_VALUE = "incompatible types: " +
                                                         "%s cannot be converted to %s type %n";
    public static final String UNASSIGNED_VAR = "using an unassigned variable: %s %n";
    public static final String REASSIGN_CONSTANT = "%s is constant.%n";
    public static final String UNINITIALIZED_CONSTANT = "constant variable must be initialized at " +
                                                        "declaration time. \n";
    public static final String INCOMPATIBLE_TYPES = "incompatible types: %s and %s%n";
    public static final String REDECLARATION = "Variable %s is already defined in the scope%n";

    // ############################## //
    // #### METHODS EXCEPTIONS #### //
    // ############################## //

    public static final String REDEFINITION = "method %s is already definedv%n";
    public static final String NOT_DEFINED = "method %s never defined %n";
    public static final String NESTED_METHOD = "nested methods not allowed: in method %s";
    public static final String DIFFERENT_ARGS = "in method %s: " +
                                                "actual and formal argument lists differ in length. %n";
    public static final String NO_RETURN = "method %s had no return statement. %n";
    public static final String SAME_PARAMETERS = "method %s has parameters with the same name. %n";

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * constructs an new exception
     * @param errorMsg informative error message
     */
    public VerifierExceptions(String errorMsg) {
        super(errorMsg);
    }
}
```

7 oop/ex6/main/Sjavac.java

```java
package oop.ex6.main;

import oop.ex6.verifier.FileVerifier;

/**
 * Main class: runs the verifier on the source file.
 */
public class Sjavac {

    /** wrong num of args */
    private static final String WRONG_USG = "USAGE: java oop.ex6.main.Sjavac <source_file_name>";

    /** valid num of args */
    private static final int NUM_OF_ARGS = 1;

    /**
     * main method: starts the program
     * @param args input arguments
     */
    public static void main(String[] args) {
        if (args.length != NUM_OF_ARGS){
            System.err.println(WRONG_USG);
            System.exit(1);
        }
        String sourceFile = args[0];
        FileVerifier.verifyCode(sourceFile);
    }
}
```

8 oop/ex6/symbol/Block.java

```java
package oop.ex6.symbol;

import java.util.ArrayList;

/**
 * A class represents a Block of code (= method or if\while)
 */
public class Block {

    // ###################### //
    // #### ATTRIBUTES #### //
    // ###################### //

    private int line;
    private final String name;
    private final ArrayList<Variable> parameters;

    // ####################### //
    // #### CONSTRUCTORS #### //
    // ####################### //

    /**
     * constructs a new block from the given details.
     * @param name block name (null in case of if\while block)
     * @param parameters parameters of block (method arguments, or if\while conditions)
     */
    public Block(String name, ArrayList<Variable> parameters) {
        this.name = name;
        this.parameters = parameters;
    }

    /**
     * constructs a new block from the given details.
     * @param parameters parameters of block (method arguments, or if\while conditions)
     */
    public Block(ArrayList<Variable> parameters) {
        this.name = null;
        this.parameters = parameters;
    }

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * sets the starting line of the block.
     * @param line line number
     */
    public void setStartLine(int line) {
        this.line = line;
    }

    /**
     * gets line number
     * @return line number
     */
    public int lineNum() {
        return this.line;
    }
```

```
60
61      /** gets method name */
62      public String name() {
63          return name;
64      }
65
66      /** gets block parameters */
67      public ArrayList<Variable> parameters() {
68          return parameters;
69      }
70
71      /** gets arguments number */
72      public int argsNum(){
73          return parameters.size();
74      }
75
76  }
```

9 oop/ex6/symbol/Variable.java

```java
package oop.ex6.symbol;

/**
 * A class represents a variable
 */
public class Variable{

    public static final String BOOLEAN_TYPE = "boolean";

    // ###################### //
    // #### ATTRIBUTES #### //
    // ###################### //

    private final String name;
    private final String type;
    private final String value;
    private final boolean constant;
    private boolean isArg = false;
    private boolean isCondition = false;

    // ######################## //
    // #### CONSTRUCTORS #### //
    // ######################## //

    /**
     * Constructs a new variable from the given details.
     * @param name variable name
     * @param type variable type
     * @param value variable value
     * @param constant is variable final?
     */
    public Variable(String name, String type, String value, boolean constant) {
        this.name = name;
        this.type = type;
        this.value = value;
        this.constant = constant;
    }

    /**
     * Constructs a new variable from name.
     * @param name variable name
     */
    public Variable(String name) {
        this(name, BOOLEAN_TYPE, null, false);
    }

    // ################## //
    // #### METHODS #### //
    // ################## //

    /** gets variable name */
    public String name() {
        return name;
    }

    /** gets variable type */
    public String type() {
        return type;
    }
```

```
60
61      /** gets variable value */
62      public String value() {
63          return value;
64      }
65
66      /** gets if variable final */
67      public boolean isConstant() {
68          return constant;
69      }
70
71      /** gets if variable is a method argument */
72      public boolean isArg() {
73          return isArg;
74      }
75
76      /** marks variable as argument */
77      public void markAsArg() {
78          isArg = true;
79      }
80
81      /** gets if variable is a if\while condition */
82      public boolean isCondition() {
83          return isCondition;
84      }
85
86      /** marks variable as condition */
87      public void markAsCondition() {
88          isCondition = true;
89      }
90
91  }
```

10 oop/ex6/verifier/FileVerifier.java

```java
package oop.ex6.verifier;

import oop.ex6.VerifierExceptions;
import oop.ex6.FileReader;

import java.io.IOException;

/**
 * A class combine all partial verifiers
 */
public class FileVerifier {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    /** Code check results */
    private static final int LEGAL_CODE = 0;
    private static final int ILLEGAL_CODE = 1;
    private static final int IO_ERROR_CODE = 2;

    /** IO ERROR MESSAGE */
    private static final String IO_ERR_MSG = "ERROR: IO error occurred";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * Verifies that code has no compilation errors.
     * @param sourceFile source file (.sjava) to check.
     * @return LEGAL_CODE (=0) if the code is legal,
     *           ILLEGAL_CODE (=1) if code is illegal,
     *           IO_ERROR_CODE (=2) if there is IO errors.
     */
    public static int verifyCode(String sourceFile) {
        try {
            String[] lines = new FileReader(sourceFile).getFileContent();
            SyntaxVerifier syntaxVerifier = new SyntaxVerifier(lines);
            SemanticsVerifier semanticsVerifier = new SemanticsVerifier(lines);

            syntaxVerifier.verifySyntax();
            semanticsVerifier.setupGlobalScope();
            semanticsVerifier.verifyLocalScopes();

            System.out.println(LEGAL_CODE);
            return LEGAL_CODE;
        }
        catch (IOException e){
            System.out.println(IO_ERROR_CODE);
            System.err.println(IO_ERR_MSG);
```

```
60                 return IO_ERROR_CODE;
61             }
62             catch (VerifierExceptions e){
63                 System.out.println(ILLEGAL_CODE);
64                 System.err.print(e.getMessage());
65                 return ILLEGAL_CODE;
66             }
67         }
68 
69     }
```

11 oop/ex6/verifier/SemanticsVerifier.java

```java
package oop.ex6.verifier;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.regex.Matcher;

import oop.ex6.SJavaRegex;
import oop.ex6.symbol.Block;
import oop.ex6.symbol.Variable;
import oop.ex6.VerifierExceptions;
import oop.ex6.Parser;

/**
 * A verifier class for static semantics of the code:
 * checks valid assignments and referring for variables
 */
public class SemanticsVerifier {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    private static final String ERROR = "ERROR: line %d: %n\t";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    /** All lines of the code */
    private final String [] fileLines;
    /** counter for code lines */
    private int lineNum = 1;
    /** number of blocks (method definition of if\while) in the code */
    private int blockCount = 0;
    /** code symbol table: contains all variables and methods details */
    private SymbolTable symbolTable = new SymbolTable();


    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Constructs a new Semantics Verifier
     */
    public SemanticsVerifier(String[] fileLines) {
        this.fileLines = fileLines;
    }

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * Building the global scope tables:
     * (1) methods table: contains every method details: name and parameters.
     * (2) global variables table: contains all of global variables details:
     *     constant, type, name, and value.
```

```
60          * Then it puts global variables table as the root of the LinkedList of
61          * the scopes tables.
62          *    global -> local_1 -> local_2 -> ... -> local_n
63          * @throws VerifierExceptions illegal variable or method
64          */
65         public void setupGlobalScope() throws VerifierExceptions {
66             int lineNum = 1;
67             // insert an empty global table
68             HashMap<String, Variable> varTable = new HashMap<>();
69             symbolTable.addVariableTable(varTable);
70             // iterate over line codes to catch global variables and method declarations
71             for (String line: fileLines) {
72                 // I. skip empty and comment lines
73                 if (SJavaRegex.EMPTY_LINE.matcher(line).matches() ||
74                     SJavaRegex.COMMENT_LINE.matcher(line).matches()) {
75                     ++lineNum;
76                     continue;
77                 }
78                 // II. No if\while blocks in the global scope
79                 checkGlobalIfWhile(line);
80                 // III. add method to methods table
81                 Matcher method = SJavaRegex.METHOD_DEFINITION.matcher(line);
82                 if (method.matches()) {
83                     ++blockCount;
84                     checkMethodSignature(line, lineNum);
85                     ++lineNum;
86                     continue;
87                 }
88                 // IV. add global variable(s) to table
89                 if (blockCount == 0)
90                     checkGlobalVar(line);
91                 // V. closing block
92                 Matcher closing = SJavaRegex.CLOSE_SUFFIX.matcher(line);
93                 if (closing.matches())
94                     --blockCount;
95                 ++lineNum;
96             }
97         }
98
99         /**
100          * checks if there is a global if\while block, if it is not
101          * (i.e. it is inside a method) then increment the block counter.
102          * @param line line to check
103          * @throws VerifierExceptions illegal block
104          */
105         private void checkGlobalIfWhile(String line) throws VerifierExceptions {
106             Matcher ifWhile = SJavaRegex.IF_WHILE_BLOCK.matcher(line);
107             if (ifWhile.matches()) {
108                 if (blockCount == 0) {
109                     throw new VerifierExceptions(String.format(ERROR + VerifierExceptions.GLOBAL_IF_WHILE,
110                                                                lineNum));
111                 } else {
112                     ++blockCount;
113                 }
114             }
115         }
116
117         /**
118          * check the validity of method signature, and increment the block counter.
119          * @param line line to check
120          * @throws VerifierExceptions illegal method signature
121          */
122         public void checkMethodSignature(String line, int lineNum) throws VerifierExceptions {
123             try {
124                 Parser parser = new Parser(line);
125                 Block signature = parser.parseMethodSignature();
126                 symbolTable.defineMethod(signature, lineNum);
127             } catch (VerifierExceptions methodExp) {
```

```
128             throw new VerifierExceptions(String.format(ERROR + methodExp.getMessage(), lineNum));
129         }
130     }
131 
132     /**
133      * check global variable validity
134      * @param line line to check
135      * @throws VerifierExceptions iilegal variable
136      */
137     private void checkGlobalVar(String line) throws VerifierExceptions {
138         try {
139             Matcher varDec = SJavaRegex.VARIABLE_DECLARATION.matcher(line);
140             Matcher varAssign = SJavaRegex.VARIABLE_ASSIGNMENT.matcher(line);
141             if (!varDec.matches() && !varAssign.matches()) {
142                 throw new VerifierExceptions(String.format(VerifierExceptions.BAD_SYNTAX_MSG, line));
143             }
144             addLineVariables(line);
145         } catch (VerifierExceptions varExp) {
146             throw new VerifierExceptions(String.format(ERROR + varExp.getMessage(), lineNum));
147         }
148     }
149 
150     /**
151      * add variables of the given line to the table
152      * @param line line to check
153      * @throws VerifierExceptions illegal variable
154      */
155     private void addLineVariables(String line) throws VerifierExceptions {
156         Parser parser = new Parser(line);
157         ArrayList<String[]> lineVars = parser.parseVarLine();
158         for (String[] varDetail: lineVars) {
159             symbolTable.defineVariable(varDetail[0], varDetail[1],
160                                        varDetail[2], varDetail[3]);
161         }
162     }
163 
164     /**
165      * verify all of the local scopes (all methods and if\while blocks in the code)
166      * @throws VerifierExceptions illegal line
167      */
168     public void verifyLocalScopes() throws VerifierExceptions {
169         for (Block block : symbolTable.getMethodsTable().values()) {
170             int endLine = verifyBlock(block);
171             /* iterate reversely over lines until the first not empty or comment line,
172              * if it is not the return statement, then throw exception */
173             for (int idx = endLine-1; idx >= block.lineNum() - 1; --idx) {
174                 Matcher empty = SJavaRegex.EMPTY_LINE.matcher(fileLines[idx]);
175                 Matcher comment = SJavaRegex.COMMENT_LINE.matcher(fileLines[idx]);
176                 if (empty.matches() || comment.matches())
177                     continue;
178                 Matcher returnMatcher = SJavaRegex.RETURN_STATEMENT.matcher(fileLines[idx]);
179                 if (returnMatcher.matches())
180                     break;
181                 throw new VerifierExceptions(String.format(ERROR + VerifierExceptions.NO_RETURN,
182                                                            endLine, block.name()));
183             }
184         }
185     }
186 
187     /**
188      * verify a single method and its inner blocks.
189      * @param block block to verify.
190      * @return line number the block ends at.
191      * @throws VerifierExceptions illegal block
192      */
193     private int verifyBlock(Block block) throws VerifierExceptions {
194 
195         /* start building the local symbol table, which is a variable
```

```
196             * table because no nested method definition */
197             HashMap<String, Variable> locVarTable = new HashMap<>();
198             symbolTable.addVariableTable(locVarTable);
199             // add args to local table, all are (final)? declared unassigned variables
200             for (Variable arg: block.parameters()) {
201                 locVarTable.put(arg.name(), arg);
202             }
203             int lineNum = block.lineNum();
204             try {
205                 while (!SJavaRegex.CLOSE_SUFFIX.matcher(fileLines[lineNum]).matches()) {
206                     String line = fileLines[lineNum];
207                     Matcher methodMatcher = SJavaRegex.METHOD_DEFINITION.matcher(line);
208                     // I. defining method in other than global scope is illegal
209                     if (methodMatcher.matches() && blockCount != 0) {
210                         throw new VerifierExceptions(String.format(ERROR + VerifierExceptions.NESTED_METHOD,
211                                                                    lineNum+1,
212                                                                    methodMatcher.group("name")));
213                     }
214                     // II. add variables to local table, update previous tables if needed
215                     else if (SJavaRegex.VARIABLE_DECLARATION.matcher(line).matches() ||
216                              SJavaRegex.VARIABLE_ASSIGNMENT.matcher(line).matches())
217                         addLineVariables(line);
218                     // III. verify method call
219                     else if (SJavaRegex.METHOD_CALL.matcher(line).matches())
220                         verifyMethodCall(line);
221                     // IV. verify if\while line and recursively verify its block
222                     if (SJavaRegex.IF_WHILE_BLOCK.matcher(line).matches()) {
223                         Block ifWhileBlock = verifyIfWhile(line);
224                         ifWhileBlock.setStartLine(lineNum + 1);
225                         lineNum = verifyBlock(ifWhileBlock);
226                     }
227                     ++lineNum;
228                 }
229                 symbolTable.removeBlockTable();
230                 return lineNum;
231             } catch (VerifierExceptions e) {
232                 throw new VerifierExceptions(String.format(ERROR + e.getMessage(), lineNum+1));
233             }
234         }
235
236         /**
237          * check method call line = name (args*)
238          * @param line code line
239          * @throws VerifierExceptions illegal method call
240          */
241         private void verifyMethodCall(String line) throws VerifierExceptions {
242             Parser parser = new Parser(line);
243             ArrayList<String> lineDetails = parser.parseCallLine();
244             Iterator<String> calledMethodDetails = lineDetails.iterator();
245             String methodName = calledMethodDetails.next();
246             // check if method is defined
247             if (!symbolTable.getMethodsTable().containsKey(methodName))
248                 throw new VerifierExceptions(String.format(VerifierExceptions.NOT_DEFINED, methodName));
249             Block block = symbolTable.getMethodsTable().get(methodName);
250             // check number of parameters
251             if (lineDetails.size() - 1 != block.argsNum())
252                 throw new VerifierExceptions(String.format(VerifierExceptions.DIFFERENT_ARGS, methodName));
253             // check parameters compatibility
254             Iterator<Variable> definedVars = block.parameters().iterator();
255             while (calledMethodDetails.hasNext() && definedVars.hasNext()) {
256                 String definedArgType = definedVars.next().type();
257                 String passedArg = calledMethodDetails.next();
258                 symbolTable.checkVarCompatibility(definedArgType, passedArg);
259             }
260         }
261
262         /**
263          * check if\while line = if\while (conditions*)
```

```
264         * @param line code line
265         * @return if\while verified block
266         * @throws VerifierExceptions illegal if\while line
267         */
268        public Block verifyIfWhile(String line) throws VerifierExceptions {
269            Parser parser = new Parser(line);
270            ArrayList<Variable> arguments = new ArrayList<>();
271            // check parameters compatibility
272            for (String condition : parser.parseIfWhileLine()) {
273                symbolTable.checkVarCompatibility("boolean", condition);
274                if (!condition.matches(SJavaRegex.BOOL_VALUE)) {
275                    Variable conditionVar = new Variable(condition);
276                    conditionVar.markAsCondition();
277                    arguments.add(conditionVar);
278                }
279            }
280            return new Block(arguments);
281        }
282    }
```

12 oop/ex6/verifier/SymbolTable.java

```java
package oop.ex6.verifier;

import oop.ex6.SJavaRegex;
import oop.ex6.VerifierExceptions;
import oop.ex6.symbol.Block;
import oop.ex6.symbol.Variable;

import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.regex.Pattern;

/**
 * Class of symbol tables of the code
 */
public class SymbolTable {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    private static final String CONSTANT = "final";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    /** symbol table for code methods
     *  <name: final, type, value> */
    private HashMap<String, Block> methodsTable = new HashMap<>();

    /** linked list of all variable tables (for each scope there is a table) */
    private LinkedList<HashMap<String, Variable>> variablesTables = new LinkedList<>();

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    // ################## //
    // #### METHODS #### //
    // ################## //

    /** get the method table */
    public HashMap<String, Block> getMethodsTable() {
        return methodsTable;
    }

    /**
     * add a new symbol table of the current scope
     * @param table variable symbol table
     */
    public void addVariableTable(HashMap<String, Variable> table) {
        variablesTables.add(table);
    }

    /** remove the last variable table (of last scope) */
    public void removeBlockTable() {
        variablesTables.removeLast();
    }
```

```
60
61      /**
62       * find the most inner scope the variable is declared in
63       * @param name variable name to search
64       * @return A variable table if found, else null
65       */
66      public HashMap<String, Variable> variableScope(String name) {
67          Iterator<HashMap<String, Variable>> iter = variablesTables.descendingIterator();
68          while (iter.hasNext()) {
69              HashMap<String, Variable> table = iter.next();
70              if (table.containsKey(name)) {
71                  if (!table.get(name).isCondition() || table.get(name).value() != null)
72                      return table;
73              }
74          }
75          return null;
76      }
77
78      /**
79       * Defines a variable with the given details, that is, adds it to
80       * the suitable scope table, if it is a valid variable.
81       * @param name variable's name
82       * @param constant final variable or not
83       * @param type variable's type
84       * @param value variable's value
85       * @throws VerifierExceptions illegal variable
86       */
87      public void defineVariable(String name, String constant, String type, String value)
88              throws VerifierExceptions {
89          HashMap<String, Variable> currentTable = variablesTables.getLast();
90          // I. variable declaration : (final)? type name (= val)?;
91          if (type != null) {
92              boolean constantVar = CONSTANT.equals(constant);
93              // I.a. check redeclaration in the current scope
94              if (currentTable.containsKey(name))
95                  throw new VerifierExceptions(String.format(VerifierExceptions.REDECLARATION, name));
96              // I.b. check initializing constant variable = final type name;
97              if (constantVar && value == null)
98                  throw new VerifierExceptions(VerifierExceptions.UNINITIALIZED_CONSTANT);
99              // I.c. check valid assigning: type name = val
100             if (value != null)
101                 checkVarCompatibility(type, value);
102
103             currentTable.put(name, new Variable(name, type, value, constantVar));
104         }
105         // II. assignment of a variable : name = val;
106         else {
107             // II.a. check previous declaration
108             HashMap<String, Variable> scope = variableScope(name);
109             if (scope == null) {
110                 throw new VerifierExceptions(String.format(VerifierExceptions.BAD_REFERENCE, name));
111             }
112             Variable currentVar = scope.get(name);
113             // II.b check reassigning a constant variable
114             if (currentVar.isConstant()) {
115                 throw new VerifierExceptions(String.format(VerifierExceptions.REASSIGN_CONSTANT, name));
116             }
117             // II.c. check valid assigning: type name = val
118             checkVarCompatibility(currentVar.type(), value);
119
120             Variable newVar = new Variable(name, currentVar.type(), value, currentVar.isConstant());
121             currentTable.put(name, newVar);
122         }
123     }
124
125     /**
126      * defines a new method in the global scope (adds it to methods symbol table)
127      * @param blockSignature method signature
```

```
128         * @param line the line the methods starts at
129         * @throws VerifierExceptions illegal method signature.
130         */
131        public void defineMethod(Block blockSignature, int line) throws VerifierExceptions{
132            if (methodsTable.containsKey(blockSignature.name())) {
133                throw new VerifierExceptions(String.format(VerifierExceptions.REDEFINITION,
134                                                           blockSignature.name()));
135            }
136            blockSignature.setStartLine(line);
137            methodsTable.put(blockSignature.name(), blockSignature);
138        }
139
140        /**
141         * check if the a variable of the given type compatible a given value
142         * @param type type of variable
143         * @param value a given value (may be a name ot other variable)
144         * @throws VerifierExceptions illegal variable
145         */
146        public void checkVarCompatibility(String type, String value) throws VerifierExceptions {
147            // I. value is a variable name: check assignment, and type
148            if (Pattern.compile(SJavaRegex.VARIABLE_NAME).matcher(value).matches() &&
149                !value.matches("true|false")) {
150                HashMap<String, Variable> scope = variableScope(value);
151                if (scope == null || scope.get(value).value() == null && !scope.get(value).isArg()) {
152                    throw new VerifierExceptions(String.format(VerifierExceptions.UNASSIGNED_VAR, value));
153                }
154                if (!isCompatibleTypes(type, scope.get(value).type())) {
155                    throw new VerifierExceptions(String.format(VerifierExceptions.INCOMPATIBLE_TYPES,
156                                                               type, scope.get(value).type()));
157                }
158            }
159            // II. value is not a variable name:
160            else if (!isCompatibleTypeValue(type, value)) {
161                throw new VerifierExceptions(String.format(VerifierExceptions.INCOMPATIBLE_TYPE_VALUE,
162                                                           value, type));
163            }
164        }
165
166        /**
167         * check if the given types ar compatible.
168         * @param type1 first type
169         * @param type2 second type
170         * @return true if compatible, false else.
171         */
172        public boolean isCompatibleTypes(String type1, String type2) {
173            return type1.equals(type2) || (type1.equals("double") && type2.equals("int")) ||
174                    type1.equals("boolean") && ((type2.equals("int") || type2.equals("double")));
175        }
176
177        /**
178         * if the given value (not a variable name) compatible with the given type
179         * @param type variable type
180         * @param value some value
181         * @return true if compatible, false else.
182         */
183        private boolean isCompatibleTypeValue(String type, String value) {
184            switch (type) {
185            case "int":
186                return Pattern.compile(SJavaRegex.INT_VALUE).matcher(value).matches();
187            case "double":
188                return Pattern.compile(SJavaRegex.DOUBLE_VALUE).matcher(value).matches();
189            case "boolean":
190                return Pattern.compile(SJavaRegex.BOOL_VALUE).matcher(value).matches();
191            case "char":
192                return Pattern.compile(SJavaRegex.CHAR_VALUE).matcher(value).matches();
193            case "String":
194                return Pattern.compile(SJavaRegex.STRING_VALUE).matcher(value).matches();
195            default:
```

```
196                 return false;
197             }
198         }
199 
200     }
```

13 oop/ex6/verifier/SyntaxVerifier.java

```java
package oop.ex6.verifier;

import oop.ex6.SJavaRegex;
import oop.ex6.VerifierExceptions;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

/** a class verities syntax correctness */
public class SyntaxVerifier {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    private static final String ERROR = "ERROR: line %d: %n\t";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### /
    /** All lines of the code */
    private final String [] fileLines;

    /** code blocks (method definition ot if\while) */
    private int blocks = 0;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Constructs a new Semantics Verifier
     * @param fileLines all lines of the code file
     */
    public SyntaxVerifier(String[] fileLines) {
        this.fileLines = fileLines;
    }

    // ################# //
    // #### METHODS #### //
    // ################# //

    /**
     * checks line syntax validity.
     * @param line code line to check.
     * @return true if valid, else false.
     */
    private boolean isValidLine(String line) {
        for (Pattern pattern: SJavaRegex.LEGAL_PATTERN) {
            Matcher matcher = pattern.matcher(line);
            if (matcher.matches()) {
                if (pattern == SJavaRegex.METHOD_DEFINITION ||
                    pattern == SJavaRegex.IF_WHILE_BLOCK)
                    ++blocks;
                else if (pattern == SJavaRegex.CLOSE_SUFFIX)
                    --blocks;
                return true;
            }
        }
```

```
60          return false;
61      }
62
63      /**
64       * verifying the code syntax
65       * @throws VerifierExceptions bad syntax
66       */
67      public void verifySyntax() throws VerifierExceptions {
68          int lineNum = 1;
69          for (String line: fileLines) {
70              if (!isValidLine(line)) {
71                  throw new VerifierExceptions(String.format(ERROR + VerifierExceptions.BAD_SYNTAX_MSG,
72                                                             lineNum, line));
73              }
74              else if (blocks < 0)
75                  throw new VerifierExceptions(String.format(ERROR + VerifierExceptions.BAD_BRACKETS,
76                                                             lineNum));
77              ++lineNum;
78          }
79          if (blocks != 0) {
80              throw new VerifierExceptions(VerifierExceptions.BAD_BLOCKS_MSG);
81          }
82      }
83  }
```