Contents

-1 a package for exceptions is wrong... for example, excpetions for methods should be in dataStructure package

1 Basic Test Results

```
Logins: kobi_atiya

###################################################
###########  No errors found :-)!!!     ###########
###################################################
```

2 README

```
kobi_atiya

FILES LIST

..main package..
Sjava.java - This class performs the validation action on the given s-java code file

..dataStructures package..
Variable.java - A class that represents a single variable Containing the variable's type,
    value and isFinal properties
VariableMap.java - This class represents a map of variables by their name as the key
Method.java - This class represents a single method, Containing it's return type and a list
    of its' variables.
MethodMap.java - This class represents a Map of methods by the method name as the key
RETURN_TYPES.java - An Enum representing the different known return types

..scanner package..
CodeFileValidator.java - The class validates every line in the code file,
    and parses the file accordingly
CodeScanner.java - A class that wraps a Scanner and scans a s-java file
Config.java - A configuration class containing all different constants used

..exceptions package..
CodeFileNotFoundException.java - Thrown in case that the code file is not found

CompilationException.java - A general exception that all different exceptions extend
--DuplicateKeyException.java - Thrown in case of a duplicate key in the method/variable map
--FileScanException.java - Thrown when there is an error while scanning the code file
--FinalAssignmentException.java - Thrown in case of assigning a final variable after it was declared
--InvalidArgumentException.java - Thrown in case of an invalid non specific argument
--InvalidMethodNameException.java - Thrown when the name of the method is not as defined
--InvalidReturnLineException.java - Thrown when the return line is not as expected
--InvalidReturnTypeException.java - Thrown when the return type does not match the one defined in the method
--InvalidVarNameExcepetion.java - Thrown when the name of the variable is not as defined
--InvalidVarValueException.java - Thrown when the variable value does not match its' type
--NullReferenceException.java - Thrown when trying to set a variable value from another uninitialized variable
--UnknownLineException.java - Thrown in any case of a code line that is not expected

ERROR HANDLING

All exceptions except CodeFileNotFoundException extend the general
CompilationException, so that in the main method @ Sjavac.java a compilation exception
is caught and thereby catching all types of compilation exception.

Also, error handling is being used when trying to parse a given condition,
so that first we try to parse the condition as a value if an exception is thrown,
we search the variable in the local scope and if another exception
is thrown we search the variable as a data member.

DESIGN & IMPLEMENTATION ISSUES

..Data Structures..
I used a LinkedHashMap as main the data structure for both method and variable map,
so that the order in which methods/variables are inserted is maintained.
In this way it is easy to separate the input and local variables.
                                                                        good
The method map contains all methods found in the code file,
and every method contains a variable map which contains
all the variable of the method.
```

```
In that way the data members are treated as variables inside a "main" method.

..Regex & Capturing..
All regex used are capturing the known templates of lines in the code file,
Then I extract the groups from the matched string,
and only then these strings are being validated.

..File scanning..
I created the CodeScanner class that wraps a Scanner,
by using this I got all the scanning abilities of the scanner
and adjusted them to scan the code file.
Such as ignoring comments, moving to certain parts of the file, etc.
The scanning is done so that the first scan gets all the methods and data members
and only then parse and validate every code line in each method.

OOP SECTION

..Addition of a new variable type..
In order to add a new variable type, two classes needs to be changed :
The RETURN_TYPES enum, and the Variable class so that the s-java compiler
will recognize the new variable type.

..Support of multiple files compilation..
In order to support this the basic changes are :
create a regex for "import" line and a parsing method needs to CodeFileValidator class.
When an import line will match the method map of the imported file should
be copied to the current code file method map and then the method calls
will be valid.

..Two main regex used..
1. a regex for capturing the condition inside an if/while statement :
^\\s*(?:if|while)\\s*\\((.*)\\)\\s*\\{

after matching the regex with a if/while line
the group(1) will contain the condition inside the if/while statement.

2. a regex for validating that a method's name is valid
^(?!_)[A-Za-z_]\\w*

in case of a method name that is not matched by this regex,
an InvalidMethodNameException will be thrown.
```

3 oop/ex2/dataStructures/Method.java

```java
package oop.ex2.dataStructures;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;

import oop.ex2.exceptions.DuplicateKeyException;
import oop.ex2.exceptions.FinalAssignmentException;
import oop.ex2.exceptions.InvalidArgumentException;
import oop.ex2.exceptions.InvalidVarNameExcepetion;
import oop.ex2.exceptions.InvalidVarValueException;
import oop.ex2.exceptions.InvalidReturnTypeException;
import oop.ex2.exceptions.NullReferenceException;
import oop.ex2.exceptions.UnknownLineException;
import oop.ex2.scanner.Config;

/**
 * This class represents a single method, Containing it's return type and a list
 * of its' variables.
 *
 * @author kobi_atiya 301143244 kobi atiya
 */

public class Method {

    private RETURN_TYPES returnType;
    private VariableMap methodParams;

    /*Input parameters count is used to separate between local variable
     *defined inside the method and the methods input variables */
    private int inputParameterCount = 0;

    /**
     * A constructor for the Method class
     *
     * @param returnType the method's return type
     * @param parameters a string representing the input parameters of the
     * method
     * @throws FinalAssignmentException in case of creating unassigned final
     * @throws InvalidReturnTypeException in case return type is not one the
     * valid return types
     * @throws InvalidVarValueException in case variable value does not match
     * it's given type
     * @throws DuplicateKeyException in case that a variable (in this method)
     * with the same name already exists
     */
    public Method(RETURN_TYPES returnType, String parameters) throws InvalidArgumentException,
            DuplicateKeyException, InvalidVarValueException, InvalidReturnTypeException,
            FinalAssignmentException {
        this.returnType = returnType;
        this.methodParams = new VariableMap();
        if (parameters.length() == 0)
            return;
        String[] paramsArray = parameters.split(",");
        for (String param : paramsArray) {
            param = param.trim().replaceAll(Config.SPACES_CHARS, " ");
            switch (param.split(" ").length) {
            // create variable found in method declaration
            case 2:
```

```
                methodParams.createVariable(param.split(" ")[0], param.split(" ")[1], false);
                inputParameterCount++;
                break;
            // variable in declaration is final
            case 3:
                if (param.split(" ")[0].equals("final"))
                    methodParams.createVariable(param.split(" ")[1], param.split(" ")[2],
                            Variable.parseFinal(param.split(" ")[0]));
                else
                    throw new InvalidArgumentException();
                inputParameterCount++;
                break;
            //throw an exception in case of too many ","
            default:
                throw new InvalidArgumentException();
            }
        }
    }

    /**
     * A copy constructor for the method class Copies the return type and
     * variable to this method from other method
     *
     * @param other the other method to copy variables from
     */

    public Method(Method other) {
        this.returnType = other.returnType;
        this.methodParams = new VariableMap(other.getVarMap());
        this.inputParameterCount = other.inputParameterCount;
    }

    private VariableMap getVarMap() {
        return new VariableMap(methodParams);
    }

    /**
     * Add a new variable for the current method
     *
     * @param name variable name
     * @param type variable type
     * @param value a string representing the variable value
     * @param isFinal a boolean that determines if the variable is final
     * @throws InvalidReturnTypeException in case return type is not one the
     * valid return types
     * @throws FinalAssignmentException in case of creating unassigned final
     * @throws InvalidVarNameExcepetion in case the the variable name is not
     * valid
     * @throws InvalidVarValueException in case variable value does not match
     * it's given type
     * @throws NullReferenceException in case that the variable we copy from is not assigned
     */
    public void createVariable(String name, String type, String value, boolean isFinal)
            throws DuplicateKeyException, InvalidReturnTypeException, FinalAssignmentException,
            InvalidVarNameExcepetion, InvalidVarValueException, NullReferenceException {
        try {
            methodParams.createVariable(type, name, value, isFinal);
        } catch (InvalidVarValueException e) {
            //In case that the given value is another variable name,
            //copy the value from the other var and create a new one with his value
            methodParams.createVariableFromOther(type, name, value, isFinal);
        }
    }

    /**
     * Add a new variable for the current method
     *
     * @param name variable name
```

```
128         * @param type variable type
129         * @param isFinal isFinal a boolean that determines if the variable is final
130         * @throws FinalAssignmentException in case of creating unassigned final
131         * @throws InvalidReturnTypeException in case return type is not one the
132         * valid return types
133         * @throws InvalidVarValueException in case variable value does not match
134         * it's given type
135         * @throws DuplicateKeyException in case that the a variable with the same
136         * already exists
137         */
138
139        public void createVariable(String name, String type, boolean isFinal)
140                throws DuplicateKeyException, InvalidVarValueException, InvalidReturnTypeException,
141                FinalAssignmentException {
142            methodParams.createVariable(type, name, null, isFinal);
143        }
144
145        /**
146         * Returns the value of a given varialbe in the map
147         *
148         * @param name the name of the variable of which we get the value of
149         * @return the value of a given varialbe in the map
150         * @throws InvalidVarNameExcepetion in case variable value does not match
151         * it's given type
152         */
153
154        public String getVariableValue(String name) throws InvalidVarNameExcepetion {
155            return methodParams.getValue(name);
156        }
157
158        /**
159         * Sets a variable value for this method
160         *
161         * @param name the variable name
162         * @param value the new value of the variable
163         * @throws FinalAssignmentException in case of creating unassigned final
164         * @throws InvalidVarNameExcepetion in case the the variable name is not
165         * valid
166         * @throws InvalidVarValueException in case variable value does not match
167         * it's given type
168         */
169        public void setVariable(String name, String value) throws InvalidVarNameExcepetion,
170                FinalAssignmentException, InvalidVarValueException {
171            try {
172                methodParams.setValue(name, value);
173            } catch (InvalidVarValueException e) {
174                methodParams.setValueFromOther(name, value);
175            }
176        }
177
178        /**
179         * Gets the return type of the method
180         *
181         * @return an Enum representing the return type of the method
182         */
183        public RETURN_TYPES getReturnType() {
184            return (returnType);
185        }
186
187        /**
188         * Gets the number of input parameters
189         *
190         * @return number of input parameters
191         */
192        public int getInputParametersCount() {
193            return (inputParameterCount);
194        }
195
```

```
196     /**
197      * Get a set of the method's variables names
198      *
199      * @return a set of the method's variables names
200      */
201     public Set<String> getVarNames() {
202         return (methodParams.getVarNames());
203     }
204 
205     /**
206      * Get a list of strings which are the keys of this method map
207      *
208      * @return a list of strings which are the keys of this method map
209      */
210 
211     public List<String> getKeyList() {
212         List<String> list = new ArrayList<String>();
213         list.addAll(methodParams.getKeyList());
214         return list;
215     }
216 
217     /**
218      * Gets the type of the given variable name
219      *
220      * @param variableName the name of the variable
221      * @return An Enum representing the type of the variable
222      * @throws InvalidVarNameExcepetion in case the the variable name is not
223      * valid
224      */
225     public RETURN_TYPES getVariableReturnType(String variableName) throws InvalidVarNameExcepetion {
226         return (methodParams.getType(variableName));
227     }
228 
229     public String toString() {
230         String result = "";
231         return result += "ReturnType: " + getReturnType().toString() + "\n"
232                 + methodParams.toString();
233     }
234 
235     /**
236      * Returns true if the methods' return type is boolean/int/double, false
237      * otherwise
238      *
239      * @return true if the methods' return type is boolean/int/double, false
240      * otherwise
241      */
242 
243     public boolean returnsBool() {
244         if (returnType.equals(RETURN_TYPES.BOOLEAN) || returnType.equals(RETURN_TYPES.INT)
245                 || returnType.equals(RETURN_TYPES.DOUBLE))
246             return true;
247         return false;
248 
249     }
250 
251     /**
252      * Splits a condition inside if/while/return to a list of strings handles
253      * complex method calls inside the condition
254      *
255      * @param string the condition to split
256      * @return an array of strings which are the sub conditions of the given string
257      * @throws UnknownLineException in case given string is invalid
258      */
259 
260     public static String[] splitCondition(String string) throws UnknownLineException {
261         ArrayList<String> resultStrings = new ArrayList<String>();
262         String temp = "";
263         /*
```

```
264             * In order to parse the condition, the normal split by the "," char returned
265             * a bad result. For example for this condition : "(foo(a,b))"
266             * normal split would result in two strings "(foo(a" and "b))"
267             * while this split will return a single string of (foo(a,b)).
268             * It counts the brackets and brings the correct result,
269             * so that the inner condition can be validated recursively
270             */
271            char[] charArray = string.toCharArray();
272            for (int i = 0; i < charArray.length; i++) {
273                if (charArray[i] == ',') {
274                    resultStrings.add(temp);
275                    temp = "";
276                } else if (charArray[i] == '(') {
277                    int bracktesCounter = 1;
278                    temp = temp + charArray[i];
279                    i++;
280                    try {
281                        while (bracktesCounter > 0) {
282                            if (charArray[i] == '('
283                                bracktesCounter++;
284                            if (charArray[i] == ')')
285                                bracktesCounter--;
286                            temp = temp + charArray[i];
287                            if (bracktesCounter > 0)
288                                i++;
289                        }
290                    } catch (ArrayIndexOutOfBoundsException e) {
291                        throw new UnknownLineException(string);
292                    }
293                } else {
294                    temp = temp + charArray[i];
295                }
296            }
297            resultStrings.add(temp);
298
299            //Copy back the list to an array of strings
300            String[] resultArray = new String[resultStrings.size()];
301            for (String str : resultStrings) {
302                resultArray[resultStrings.indexOf(str)] = str;
303            }
304            return resultArray;
305        }
306    }
```

-1.5 magic numbers

4 oop/ex2/dataStructures/MethodMap.java

```java
package oop.ex2.dataStructures;

import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;

import oop.ex2.exceptions.DuplicateKeyException;
import oop.ex2.exceptions.FinalAssignmentException;
import oop.ex2.exceptions.InvalidArgumentException;
import oop.ex2.exceptions.InvalidMethodNameException;
import oop.ex2.exceptions.InvalidVarNameExcepetion;
import oop.ex2.exceptions.InvalidReturnTypeException;
import oop.ex2.exceptions.InvalidVarValueException;
import oop.ex2.exceptions.NullReferenceException;

/**
 * This class represents a Map of methods by the method name as the key
 *
 * @author kobi_atiya 301143244 kobi atiya
 */
public class MethodMap {
    private LinkedHashMap<String, Method> methodsMap;

    /**
     * A constructor for the MethodMap class
     */
    public MethodMap() {
        methodsMap = new LinkedHashMap<String, Method>();
    }

    /**
     * Adds a new method to the method map
     *
     * @param name the name of the method
     * @param returnType the return type of the method
     * @param params a string representing the methods parameters
     * @throws InvalidArgumentException in case method's input parameters string
     * is invalid
     * @throws DuplicateKeyException in case that a method with the same name is
     * already in the map
     * @throws FinalAssignmentException in case of creating unassigned final
     * @throws InvalidReturnTypeException in case return type is not one the
     * valid return types
     * @throws InvalidVarValueException in case variable value does not match
     * it's given type
     */

    public void addMethod(String name, String returnType, String params)
            throws InvalidArgumentException, DuplicateKeyException, InvalidVarValueException,
            InvalidReturnTypeException, FinalAssignmentException {
        // Cannot have two methods with the same name
        if (methodsMap.containsKey(name))
            throw new DuplicateKeyException(name);
        methodsMap.put(name, new Method(Variable.parseVarType(returnType), params));
    }

    /**
     * Adds a new variable to a method in the method map
```

```
60          *
61          * @param methodName the name of the method to add the variable to
62          * @param variableName the name of the variable to add to the method
63          * @param type the variable type
64          * @param value a string representing the variable value
65          * @param isFinal a boolean that determines if the variable is final
66          * @throws InvalidVarNameExcepetion in case that the variable name is not
67          * valid
68          * @throws FinalAssignmentException in case of creating unassigned final
69          * @throws InvalidReturnTypeException in case return type is not one the
70          * valid return types
71          * @throws DuplicateKeyException in case a variable with same name already
72          * exists in the given method
73          * @throws InvalidVarValueException in case variable value does not match
74          * it's given type
75          * @throws NullReferenceException in case copy from variable is null
76          */
77         public void addVariable(String methodName, String variableName, String type, String value,
78                 Boolean isFinal) throws DuplicateKeyException, InvalidReturnTypeException,
79                 FinalAssignmentException, InvalidVarNameExcepetion, InvalidVarValueException,
80                 NullReferenceException {
81             methodsMap.get(methodName).createVariable(variableName, type, value, isFinal);
82         }
83
84         /**
85          * Adds a new variable to a method in the method map
86          *
87          * @param methodName the name of the method to add the variable to
88          * @param variableName the name of the variable to add to the method
89          * @param type the variable type
90          * @throws NullReferenceException in case copy from variable is null
91          * @throws InvalidVarValueException in case variable value does not match
92          *             it's given type
93          * @throws InvalidVarNameExcepetion in case that the variable name is not
94          *             valid
95          * @throws FinalAssignmentException in case of creating unassigned final
96          * @throws InvalidReturnTypeException in case return type is not one the
97          *             valid return types
98          * @throws DuplicateKeyException in case a variable with same name already
99          *             exists in the given method
100         */
101        public void addVariable(String methodName, String variableName, String type, Boolean isFinal)
102                throws DuplicateKeyException, InvalidReturnTypeException, FinalAssignmentException,
103                InvalidVarNameExcepetion, InvalidVarValueException, NullReferenceException {
104            addVariable(methodName, variableName, type, null, isFinal);
105        }
106
107        /**
108         * Checks if the method map contains the given method name
109         *
110         * @param name the name of method to check
111         * @return true if the method is in the map, false otherwise
112         */
113        public boolean containsMethod(String name) {
114            return (methodsMap.containsKey(name));
115        }
116
117        /**
118         * Returns the method by the given name
119         *
120         * @param name the name of the method
121         * @return a method from this method map
122         * @throws InvalidMethodNameException in case that the method is not found
123         * in the map
124         */
125
126        public Method getMethod(String name) throws InvalidMethodNameException {
127            if (methodsMap.containsKey(name)) return (methodsMap.get(name));
```

```
128             else
129                 throw new InvalidMethodNameException(name);
130         }
131 
132         /**
133          * Gets an Enum that represents the return type of the given method name
134          *
135          * @param name the name of method
136          * @return an Enum that represents the method's return type
137          * @throws InvalidVarNameExcepetion in case that the given varible is not
138          * found in the map
139          */
140         public RETURN_TYPES getReturnType(String name) throws InvalidVarNameExcepetion {
141             if (methodsMap.containsKey(name)) return (methodsMap.get(name).getReturnType());
142             throw new InvalidVarNameExcepetion();
143         }
144 
145         /**
146          * Return a string representation of the MethodMap
147          * containing the methods inside it and their variables
148          */
149 
150         public String toString() {
151             String result = "Method List:\n";
152             Set<Map.Entry<String, Method>> set = methodsMap.entrySet();
153             for (Entry<String, Method> method : set) {
154                 result += "Type: " + method.getValue().getReturnType().toString() + " Name: "
155                         + method.getKey().toString() + " Vars: " + method.getValue().getVarNames()
156                         + "\n";
157             }
158             return result;
159         }
160 
161         /**
162          * Gets a set of strings which holds the name of the methods in the map
163          *
164          * @return a set of methods names
165          */
166         public Set<String> getMethodsName() {
167             return (methodsMap.keySet());
168         }
169 
170     }
```

5 oop/ex2/dataStructures/RETURN TYPES.java

```java
package oop.ex2.dataStructures;

/**
 * An Enum representing the different known return types
 *
 * @author kobi_atiya 301143244 kobi atiya
 */
public enum RETURN_TYPES {
    INT, STRING, DOUBLE, CHAR, BOOLEAN, VOID;
}
```

6 oop/ex2/dataStructures/Variable.java

```java
package oop.ex2.dataStructures;

import oop.ex2.exceptions.FinalAssignmentException;
import oop.ex2.exceptions.InvalidArgumentException;
import oop.ex2.exceptions.InvalidVarValueException;
import oop.ex2.exceptions.InvalidReturnTypeException;

/**
 * A class that represents a single variable Containing the variable's type,
 * value and isFinal properties
 *
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class Variable {

    private RETURN_TYPES type;
    private String value;
    private boolean isFinal;

    /**
     * A constructor for the Variable class
     *
     * @param type a string representing the variable type
     * @param value the variables value
     * @param isFinal a boolean that determines if the variable is final
     * @throws InvalidVarValueException in case variable value does not match
     * it's given type
     * @throws InvalidReturnTypeException in case return type is not one the
     * valid return types
     * @throws FinalAssignmentException in case of creating unassigned final
     */
    public Variable(String type, String value, boolean isFinal) throws InvalidVarValueException,
            InvalidReturnTypeException, FinalAssignmentException {
        //In case value is final but not initialized
        if (isFinal && value == null)
            throw new FinalAssignmentException(value);
        //A variable cannot have a void return type
        if (!type.equals("void"))
            this.type = parseVarType(type);
        else
            throw new InvalidVarValueException();
        this.value = parseVariable(this.type, value);
        this.isFinal = isFinal;
    }

    /**
     * Parses a string to a variable type
     *
     * @param type the string representing the variable type
     * @return an Enum representing the variable type
     * @throws InvalidReturnTypeException in case return type is not one the
     * valid return types
     */
    public static RETURN_TYPES parseVarType(String type) throws InvalidReturnTypeException {
        if (type.equals("int"))
            return RETURN_TYPES.INT;
        if (type.equals("String"))
            return RETURN_TYPES.STRING;
```

```
60          if (type.equals("double"))
61              return RETURN_TYPES.DOUBLE;
62          if (type.equals("boolean"))
63              return RETURN_TYPES.BOOLEAN;
64          if (type.equals("char"))
65              return RETURN_TYPES.CHAR;
66          if (type.equals("void"))
67              return RETURN_TYPES.VOID;
68          throw new InvalidReturnTypeException(type);
69      }
70
71      /**
72       * Returns the is final property of this variable
73       *
74       * @return the is final property of the variable
75       */
76      public boolean isFinal() {
77          return isFinal;
78      }
79
80      /**
81       * Sets the given value to the variable
82       *
83       * @param newValue the new value to set
84       * @throws InvalidVarValueException in case variable value does not match
85       * it's given type
86       * @throws FinalAssignmentException in case of creating unassigned final
87       */
88      public void set(String newValue) throws InvalidVarValueException, FinalAssignmentException {
89          //Cannot assign a final variable
90          if (isFinal) throw new FinalAssignmentException(newValue);
91          this.value = parseVariable(type, newValue);
92      }
93
94      /**
95       * Returns the value of this variable
96       *
97       * @return the value of this variable
98       */
99      public String getValue() {
100         if (type.equals(RETURN_TYPES.STRING))
101             return ("\"" + this.value + "\"");
102         if (type.equals(RETURN_TYPES.CHAR))
103             return ("\'" + this.value + "\'");
104         return (this.value);
105     }
106
107     /**
108      * Returns the type of this variable
109      *
110      * @return the type of this variable
111      */
112     public RETURN_TYPES getType() {
113         return (this.type);
114     }
115
116     /**
117      * Parses a boolean string to a boolean The string can be "true/false" or a
118      * number
119      *
120      * @param bool the string representing a boolean value
121      * @return a string "true"/"false"
122      * @throws InvalidVarValueException in case variable value is not
123      * true/false/number
124      */
125
126     public static String parseBool(String bool) throws InvalidVarValueException {
127         if (bool.equals("false") || bool.equals("0"))
```

```
128             return "false";
129         else if (bool.equals("true") || Double.parseDouble(bool) != Double.NaN)
130             return "true";
131         throw new InvalidVarValueException(bool);
132     }
133 
134     /**
135      * Parses a final string to final property
136      *
137      * @param finalString the string expected to be final
138      * @return true if the string is "final", throws exception otherwise
139      * @throws InvalidArgumentException in case given string is not final
140      */
141 
142     public static Boolean parseFinal(String finalString) throws InvalidArgumentException {
143         if (finalString.equals("final"))
144             return true;
145         else
146             throw new InvalidArgumentException(finalString);
147     }
148 
149     /**
150      * Parses a variable according to the variable type
151      *
152      * @param type an Enum representing the type of the variable
153      * @param value the value of the variable
154      * @return a string representing the value
155      * @throws InvalidVarValueException in case variable value does not match
156      * it's given type
157      */
158     public static String parseVariable(RETURN_TYPES type, String value)
159             throws InvalidVarValueException {
160         if (value == null)
161             return null;
162         //Try parse the given value according to the expected type
163         //If parsing fails, an exception is thrown
164         switch (type) {
165         case INT:
166             try {
167                 Integer.parseInt(value);
168                 return value;
169             } catch (NumberFormatException e) {
170                 throw new InvalidVarValueException(type,value);
171             }
172         case DOUBLE:
173             try {
174                 Double.parseDouble(value);
175                 return value;
176             } catch (NumberFormatException e) {
177                 throw new InvalidVarValueException(type,value);
178             }
179         case BOOLEAN:
180             try {
181                 return (parseBool(value));
182             } catch (NumberFormatException e) {
183                 throw new InvalidVarValueException(type,value);
184             }
185         case CHAR:
186             try {
187                 value = value.substring(value.indexOf("'") + 1, value.lastIndexOf("'"));
188                 if (value.length() >= 2)
189                     throw new InvalidVarValueException();
190                 return value;
191             } catch (IndexOutOfBoundsException e) {
192                 throw new InvalidVarValueException(type,value);
193             }
194         case STRING:
195             try {
```

```
196                     value = value.substring(value.indexOf("\"") + 1, value.lastIndexOf("\""));
197                     return value;
198                 } catch (IndexOutOfBoundsException e) {
199                     throw new InvalidVarValueException(type,value);
200                 }
201             }
202             return null;
203         }
204     }
```

7 oop/ex2/dataStructures/VariableMap.java

```java
package oop.ex2.dataStructures;

import java.util.*;
import java.util.Map.Entry;

import oop.ex2.exceptions.DuplicateKeyException;
import oop.ex2.exceptions.FinalAssignmentException;
import oop.ex2.exceptions.InvalidVarNameExcepetion;
import oop.ex2.exceptions.InvalidVarValueException;
import oop.ex2.exceptions.InvalidReturnTypeException;
import oop.ex2.exceptions.NullReferenceException;
import oop.ex2.scanner.Config;

/**
 * This class represents a map of variables by their name as the key
 *
 * @author kobi_atiya 301143244 kobi atiya
 */
public class VariableMap {

    /* Contains a map of Variables where the name is the variable key
     * Used a LinkedHashMap to maintain the insertion order so that
     * input variable are stored before locals
     */
    private LinkedHashMap<String, Variable> variableMap;

    /**
     * A constructor for the VariableMap class
     */
    public VariableMap() {
        variableMap = new LinkedHashMap<String, Variable>();
    }

    /**
     * A copy constructor for the VariableMap class
     *
     * @param other the variable map to copy from
     */
    public VariableMap(VariableMap other) {
        this();
        variableMap.putAll(other.getVariableMap());
    }

    private LinkedHashMap<String, Variable> getVariableMap() {
        return this.variableMap;
    }

    /**
     * Add a new variable to the variable map
     *
     * @param type the type of the variable
     * @param name the name of the variable
     * @param value the value of the variable
     * @param isFinal a boolean that determines if the variable is final
     * @throws FinalAssignmentException in case of creating unassigned final
     * @throws InvalidReturnTypeException in case return type is not one the
     * valid return types
     * @throws InvalidVarValueException in case variable value does not match
     * it's given type
```

```
60          * @throws DuplicateKeyException in case that the variable name is already
61          * in the map
62          */
63         public void createVariable(String type, String name, String value, boolean isFinal)
64                 throws InvalidVarValueException, InvalidReturnTypeException, FinalAssignmentException,
65                 DuplicateKeyException {
66             if (!variableMap.containsKey(name)) {
67                 variableMap.put(name, new Variable(type, value, isFinal));
68             }
69             // variable with the same name already exists
70             else
71                 throw new DuplicateKeyException(name);
72         }
73 
74         /**
75          * Add a new variable to the variable map Sets default values to the varible
76          * accroding to it's type: Empty for String and Char, 0 for
77          * Boolean/Int/Double
78          *
79          * @param type the type of the variable
80          * @param name the name of the variable
81          * @param isFinal a boolean that determines if the variable is final
82          * @throws DuplicateKeyException in case that the variable name is already
83          * in the map
84          * @throws FinalAssignmentException in case of creating unassigned final
85          * @throws InvalidReturnTypeException in case return type is not one the
86          * valid return types
87          * @throws InvalidVarValueException in case variable value does not match
88          * it's given type
89          */
90         public void createVariable(String type, String name, boolean isFinal)
91                 throws DuplicateKeyException, InvalidVarValueException, InvalidReturnTypeException,
92                 FinalAssignmentException {
93 
94             String defaultValue = null;
95             //Set default values for every variable type
96             switch (Variable.parseVarType(type)) {
97             case STRING:
98                 defaultValue = Config.STRING_DEFAULT_VALUE;
99                 break;
100            case INT:
101            case DOUBLE:
102            case BOOLEAN:
103                defaultValue = Config.BOOLEAN_DEFAULT_VALUE;
104                break;
105            case CHAR:
106                defaultValue = Config.CHAR_DEFAULT_VALUE;
107                break;
108            }
109            createVariable(type, name, defaultValue, isFinal);
110        }
111 
112        /**
113         * Add a variable from another variable
114         *
115         * @param type the new variable type
116         * @param name the new variable name
117         * @param otherVariable the variable to copy the variable value from
118         * @param isFinal a boolean that determines if the variable is final
119         * @throws InvalidVarNameExcepetion in case the the variable name is not
120         * valid
121         * @throws DuplicateKeyException in case that the variable name is already
122         * in the map
123         * @throws FinalAssignmentException in case of creating unassigned final
124         * @throws InvalidReturnTypeException in case return type is not one the
125         * valid return types
126         * @throws InvalidVarValueException in case variable value does not match
127         * it's given type
```

```
128         * @throws NullReferenceException in case copy from variable has no value
129         */
130        public void createVariableFromOther(String type, String name, String otherVariable,
131                boolean isFinal) throws InvalidVarNameExcepetion, InvalidReturnTypeException,
132                FinalAssignmentException, DuplicateKeyException, InvalidVarValueException,
133                NullReferenceException {
134            //In case we try to copy a value from an unassigned variable, throw exception
135            if (getValue(otherVariable) == null)
136                throw new NullReferenceException(otherVariable);
137            createVariable(type, name, getValue(otherVariable), isFinal);
138        }
139
140        /**
141         * Gets the value of the given variable name
142         *
143         * @param name the name of the variable
144         * @return the string representing the variable value
145         * @throws InvalidVarNameExcepetion in case that the variable name is not found
146         * in the variable map
147         */
148
149        public String getValue(String name) throws InvalidVarNameExcepetion {
150            if (variableMap.containsKey(name)) {
151                return (variableMap.get(name).getValue());
152            } else
153                throw new InvalidVarNameExcepetion(name);
154
155        }
156
157        /**
158         * Get a list of strings which are the keys of this variable map
159         *
160         * @return A list of strings which are the keys of this variable map
161         */
162
163        public List<String> getKeyList() {
164            List<String> list = new ArrayList<String>();
165            list.addAll(variableMap.keySet());
166            return list;
167        }
168
169        /**
170         * Gets the variable type by the variable name
171         *
172         * @param name the name of the variable
173         * @return an Enum representing the variable type
174         * @throws InvalidVarNameExcepetion in case that the variable name is not
175         * found in the variable map
176         */
177        public RETURN_TYPES getType(String name) throws InvalidVarNameExcepetion {
178            if (variableMap.containsKey(name)) {
179                return (variableMap.get(name).getType());
180            } else
181                throw new InvalidVarNameExcepetion();
182        }
183
184        /**
185         * Sets the value of the given variable
186         *
187         * @param name the name of the variable
188         * @param value the value of the variable to set
189         * @throws FinalAssignmentException in case of setting a final variable
190         * @throws InvalidVarValueException in case variable value does not match
191         * it's given type
192         * @throws InvalidVarNameExcepetion in case that the variable name is not
193         * found in the variable map
194         */
195        public void setValue(String name, String value) throws InvalidVarNameExcepetion,
```

```
196             InvalidVarValueException, FinalAssignmentException {
197         if (variableMap.containsKey(name)) {
198             variableMap.get(name).set(value);
199         } else
200             throw new InvalidVarNameExcepetion(name);
201     }
202 
203     /**
204      * Sets a value of a variable from another variable by it's name
205      *
206      * @param name the name of the variable to change the value of
207      * @param otherVariable the name of variable we get the value from
208      * @throws FinalAssignmentException in case of setting a final variable
209      * @throws InvalidVarValueException in case variable value does not match
210      * it's given type
211      * @throws InvalidVarNameExcepetion in case that the variable name is not
212      * valid
213      */
214     public void setValueFromOther(String name, String otherVariable)
215             throws InvalidVarNameExcepetion, InvalidVarValueException, FinalAssignmentException {
216         setValue(name, getValue(otherVariable));
217     }
218 
219     /**
220      * Gets the isFinal property of the variable
221      *
222      * @param name the variable name
223      * @return a boolean telling if the variable is final
224      * @throws InvalidVarNameExcepetion in case that the variable name is not
225      * valid
226      */
227     public boolean getIsFinal(String name) throws InvalidVarNameExcepetion {
228         if (variableMap.containsKey(name))
229             return (variableMap.get(name).isFinal());
230         throw new InvalidVarNameExcepetion();
231     }
232 
233     /**
234      * Gets a set of strings containing the name of the variables in the map
235      *
236      * @return a set of strings containing the name of the variables in the map
237      */
238     public Set<String> getVarNames() {
239         return (variableMap.keySet());
240     }
241 
242     /**
243      * Tells if the map contains the given variable name
244      *
245      * @param name the name of the variable
246      * @return true if the variable is in the map, false otherwise
247      */
248     public boolean containsVariable(String name) {
249         return (variableMap.containsKey(name));
250     }
251 
252     /**
253      * Returns a string representation of the VariableMap,
254      * Containing the a list of variables and their values and types
255      */
256 
257     public String toString() {
258         String result = "Variales List: \n";
259         Set<Map.Entry<String, Variable>> set = variableMap.entrySet();
260         for (Entry<String, Variable> var : set) {
261             result += "Type: " + var.getValue().getType().toString() + " Name: "
262                     + var.getKey().toString() + " Value: " + var.getValue().getValue()
263                     + " IsFinal: " + var.getValue().isFinal() + "\n";
```

```
264             }
265             return result;
266         }
267     }
```

8 oop/ex2/exceptions/CodeFileNotFoundException.jav

```java
package oop.ex2.exceptions;

/**
 * Thrown in case that the code file is not found
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class CodeFileNotFoundException extends Exception {

    private static final long serialVersionUID = 1L;

    public CodeFileNotFoundException() {
    }

    /**
     *
     * @param message file path that was not found
     */
    public CodeFileNotFoundException(String message) {
        super("File not found :" + message);
    }

}
```

9 oop/ex2/exceptions/CompilationException.java

```java
package oop.ex2.exceptions;
/**
 * A general exception that all different exceptions extend
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class CompilationException extends Exception {


    private static final long serialVersionUID = 1L;

    public CompilationException() {
    }

    public CompilationException(String message) {
        super("CompilationException: " + message);
    }

}
```

10 oop/ex2/exceptions/DuplicateKeyException.java

```java
package oop.ex2.exceptions;
/**
 * Thrown in case of a duplicate key in the method/variable map
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class DuplicateKeyException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public DuplicateKeyException() {
    }

    public DuplicateKeyException(String message) {
        super("DuplicateKeyException: " + message);
    }

}
```

11 oop/ex2/exceptions/FileScanException.java

```java
package oop.ex2.exceptions;

/**
 * Thrown when there is an error while scanning the code file
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class FileScanException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public FileScanException() {

    }

    public FileScanException(String message) {
        super("FileScanException:" + message);
    }

}
```

12 oop/ex2/exceptions/FinalAssignmentException.java

```java
package oop.ex2.exceptions;
/**
 * Thrown in case of assigning a final variable after it was declared
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class FinalAssignmentException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public FinalAssignmentException() {
    }

    public FinalAssignmentException(String message) {
        super("FinalAssignmentException: " + message);
    }

}
```

13 oop/ex2/exceptions/InvalidArgumentException.java

```java
package oop.ex2.exceptions;
/**
 * Thrown in case of an invalid non specific argument
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class InvalidArgumentException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public InvalidArgumentException() {
    }

    public InvalidArgumentException(String message) {
        super("InvalidArgumentException: " + message);
    }

}
```

14 oop/ex2/exceptions/InvalidMethodNameException.

```java
package oop.ex2.exceptions;

/**
 * Thrown when the name of the method is not as defined
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class InvalidMethodNameException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public InvalidMethodNameException() {
    }

    public InvalidMethodNameException(String message) {
        super("InvalidMethodNameException: " + message);
    }

}
```

15 oop/ex2/exceptions/InvalidReturnLineException.jav

```java
package oop.ex2.exceptions;
/**
 * Thrown when the return line is not as expected
 * @author kobi_atiya 301143244 kobi atiya
 *
 */

public class InvalidReturnLineException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public InvalidReturnLineException() {
    }

    public InvalidReturnLineException(String message) {
        super("InvalidReturnLineException: " + message);
    }

}
```

16 oop/ex2/exceptions/InvalidReturnTypeException.ja

```java
package oop.ex2.exceptions;

/**
 * Thrown when the return type does not match the one defined in the method
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class InvalidReturnTypeException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public InvalidReturnTypeException() {
    }

    public InvalidReturnTypeException(String message) {
        super("InvalidReturnTypeException: " + message);
    }

    public InvalidReturnTypeException(String message,String expected, String actual) {
        super("InvalidReturnTypeException: " + "Message: " + message + " Expected: " + expected + " Actual: " + actual);
    }

}
```

17 oop/ex2/exceptions/InvalidVarNameExcepetion.java

```java
package oop.ex2.exceptions;

/**
 * Thrown when the name of the variable is not as defined
 * @author kobi_atiya 301143244
 *
 */
public class InvalidVarNameExcepetion extends CompilationException {

    private static final long serialVersionUID = 1L;

    public InvalidVarNameExcepetion() {
    }

    public InvalidVarNameExcepetion(String message) {
        super("InvalidReturnTypeException: " + message);
    }

}
```

18 oop/ex2/exceptions/InvalidVarValueException.java

```java
package oop.ex2.exceptions;

import oop.ex2.dataStructures.RETURN_TYPES;

/**
 * Thrown when the variable value does not match its' type
 * @author kobi_atiya 301143244
 *
 */
public class InvalidVarValueException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public InvalidVarValueException() {
    }

    public InvalidVarValueException(String message) {
        super("InvalidVariableValueException: " + message);
    }

    public InvalidVarValueException(RETURN_TYPES varType,String value) {
        super("InvalidVaribleValueException: " + varType.toString() + " " + value);
    }

}
```

19 oop/ex2/exceptions/NullReferenceException.java

```java
package oop.ex2.exceptions;

/**
 * Thrown when trying to set a variable value from another uninitialized variable
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class NullReferenceException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public NullReferenceException() {
    }

    public NullReferenceException(String message) {
        super("NullReferenceException: " + message);
    }

}
```

20 oop/ex2/exceptions/UnknownLineException.java

```java
package oop.ex2.exceptions;

/**
 * Thrown in any case of a code line that is not expected
 * @author kobi_atiya 301143244 kobi atiya
 *
 */

public class UnknownLineException extends CompilationException {

    private static final long serialVersionUID = 1L;

    public UnknownLineException() {
    }

    public UnknownLineException(String message) {
        super("UnknownLineException: " + message);
    }

}
```

21 oop/ex2/main/Sjavac.java

```java
package oop.ex2.main;

import java.io.File;

import oop.ex2.exceptions.CodeFileNotFoundException;
import oop.ex2.exceptions.CompilationException;
import oop.ex2.scanner.CodeFileValidator;
/**
 * This class performs the validation action on the given s-java code file
 * @author kobi_atiya 301143244 kobi atiya
 *
 */
public class Sjavac {
    public static void main(String[] args) {
        try {
            CodeFileValidator.Validate(new File(args[0]));
        //In case of code file is not found
        } catch (CodeFileNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(2);
        }
        //In case of any compilation error, print message and exit
        catch (CompilationException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        finally {
            CodeFileValidator.closeScanner();
            System.exit(0);
        }
    }
}
```

22 oop/ex2/scanner/CodeFileValidator.java

```java
package oop.ex2.scanner;

import java.io.File;

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

import oop.ex2.dataStructures.Method;
import oop.ex2.dataStructures.MethodMap;
import oop.ex2.dataStructures.RETURN_TYPES;
import oop.ex2.dataStructures.Variable;
import oop.ex2.exceptions.CodeFileNotFoundException;
import oop.ex2.exceptions.DuplicateKeyException;
import oop.ex2.exceptions.FileScanException;
import oop.ex2.exceptions.FinalAssignmentException;
import oop.ex2.exceptions.InvalidArgumentException;
import oop.ex2.exceptions.InvalidMethodNameException;
import oop.ex2.exceptions.InvalidReturnLineException;
import oop.ex2.exceptions.InvalidVarNameExcepetion;
import oop.ex2.exceptions.InvalidVarValueException;
import oop.ex2.exceptions.NullReferenceException;
import oop.ex2.exceptions.UnknownLineException;
import oop.ex2.exceptions.InvalidReturnTypeException;

/**
 * The class validates every line in the code file,
 * and parses the file accordingly
 * @author kobi_atiya 301143244 kobi atiya
 *
 */

public class CodeFileValidator {

    private static Method mainMethod;
    private static MethodMap methodMap;
    private static CodeScanner codeScan;
    private static String codeLine;

    /**
     * Checks that the given file is a valid s-java code file
     *
     * @param codeFile the file containing the code to validate
     * @throws CodeFileNotFoundException
     * @throws FinalAssignmentException
     * @throws InvalidReturnTypeException
     * @throws InvalidVarValueException
     * @throws DuplicateKeyException
     * @throws InvalidArgumentException
     * @throws FileScanException
     * @throws UnknownLineException
     * @throws InvalidMethodNameException
     * @throws InvalidVarNameExcepetion
     * @throws InvalidReturnLineException
     * @throws NullReferenceException
     */

    public static void Validate(File codeFile) throws CodeFileNotFoundException,
    InvalidArgumentException, DuplicateKeyException, InvalidVarValueException,
    InvalidReturnTypeException, FinalAssignmentException, FileScanException,
```

```
60      UnknownLineException, InvalidVarNameExcepetion, InvalidMethodNameException,
61      InvalidReturnLineException, NullReferenceException {
62          methodMap = new MethodMap();
63          codeScan = new CodeScanner(codeFile);
64          mainMethod = new Method(RETURN_TYPES.VOID, "");
65
66          //Gets all methods from the file by scanning the deceleration lines
67          preScanMethods();
68
69          codeScan = new CodeScanner(codeFile);
70          // Gets all data members and adds them to the data structure
71          preScanVars();
72
73          codeScan = new CodeScanner(codeFile);
74
75          // Skip to next method deceleration line
76          codeScan.nextMethodStartLine();
77          codeLine = codeScan.getLine();
78
79          for (String methodName : methodMap.getMethodsName()) {
80              validateMethod(methodName,methodMap.getMethod(methodName),codeScan);
81              validateLastMethodLine(methodMap.getMethod(methodName), codeScan.getLastLine());
82              codeScan.nextMethodStartLine();
83          }
84
85          //Closes the file when the validation ends
86          codeScan.closeFile();
87      }
88
89      /**
90       * Verifies that the last line of every method ends with a return statement
91       * @param method the method in which the verification is done
92       * @param lastLine the last of the method
93       * @throws InvalidReturnTypeException
94       * @throws InvalidVarNameExcepetion
95       * @throws InvalidArgumentException
96       * @throws InvalidReturnLineException
97       * @throws InvalidVarValueException
98       * @throws InvalidMethodNameException
99       * @throws UnknownLineException
100      */
101
102     private static void validateLastMethodLine(Method method, String lastLine) throws InvalidReturnTypeException,
103     InvalidVarNameExcepetion, InvalidArgumentException, InvalidReturnLineException,
104     InvalidVarValueException, InvalidMethodNameException, UnknownLineException {
105         if (!parseReturnStatement(lastLine, method)) {
106             throw new InvalidReturnLineException(lastLine);
107         }
108     }
109
110     /**
111      * Validates every single line in a methods code
112      * This method tries to parse the code line for known template,
113      * if it encounters an unknown line - throws an exception
114      *
115      * @param methodName the name of the method
116      * @param method method the method in which the verification is done
117      * @param scanner the scanner which scans the code file
118      * @throws FileScanException
119      * @throws InvalidReturnTypeException
120      * @throws InvalidVarNameExcepetion
121      * @throws InvalidArgumentException
122      * @throws DuplicateKeyException
123      * @throws InvalidVarValueException
124      * @throws FinalAssignmentException
125      * @throws UnknownLineException
126      * @throws InvalidMethodNameException
127      * @throws NullReferenceException
```

```
128         */
129
130         private static void validateMethod(String methodName, Method method, CodeScanner scanner) throws FileScanException,
131         InvalidReturnTypeException, InvalidVarNameExcepetion, InvalidArgumentException,
132         DuplicateKeyException, InvalidVarValueException, FinalAssignmentException,
133         UnknownLineException, InvalidMethodNameException, NullReferenceException {
134             String currentLine;
135             CodeScanner methodCode = scanner.extractBlock();
136             while (methodCode.hasNextLine()) {
137                 methodCode.nextLine();
138                 currentLine = methodCode.getLine();
139                 if (parseReturnStatement(currentLine,method)) ;
140                 else if (parseIfWhileStatement(currentLine,method)) {
141                     validateMethod(methodName, new Method(method), methodCode);
142                 } else if (parseVarNoInit(currentLine,method)) ;
143                 else if (parseVarAssignment(currentLine,method)) ;
144                 else if (parseVarInitNotFinal(currentLine,method)) ;
145                 else if (parseVarFinalInit(currentLine,method)) ;
146                 else if (parseVarMethodAssignment(currentLine,method)) ;
147                 else if (parseMethodCallWithoutAssignment(currentLine,method)) ;
148                 else
149                     throw new UnknownLineException("Unknown line " + currentLine);
150             }
151         }
152
153         /**
154          * Match a line for a method call without assignment
155          * If matches - validate that the call is valid
156          * @param currentLine the line to be matched
157          * @param method the method the line is in
158          * @return true if valid, false otherwise
159          * @throws InvalidVarNameExcepetion
160          * @throws InvalidArgumentException
161          * @throws InvalidVarValueException
162          * @throws InvalidMethodNameException
163          * @throws UnknownLineException
164          * @throws InvalidReturnTypeException
165          */
166
167         private static boolean parseMethodCallWithoutAssignment(String currentLine, Method method)
168         throws InvalidVarNameExcepetion, InvalidArgumentException, InvalidVarValueException, InvalidMethodNameException
169         , UnknownLineException, InvalidReturnTypeException {
170             Matcher methodCallMatcher = Pattern.compile(Config.BRACKETS).matcher(currentLine);
171
172             //Use find to match next occurrence and not match the whole line
173             if (methodCallMatcher.find()) {
174                 String methodName = methodCallMatcher.group(1);
175                 validateCondition(methodMap.getReturnType(methodName), currentLine, methodMap.getMethod(methodName));
176                 return true;
177             }
178             return false;
179         }
180
181         /**
182          * Match a line for a variable assignment with value/another variable assignment
183          * If matches - validates and adds the variable to the map
184          * @param currentLine the line to be matched
185          * @param method the method the line is in
186          * @return true if valid, false otherwise
187          * @throws FinalAssignmentException
188          * @throws InvalidVarValueException
189          * @throws InvalidVarNameExcepetion
190          */
191
192         private static boolean parseVarAssignment(String currentLine,Method method) throws FinalAssignmentException,
193         InvalidVarValueException, InvalidVarNameExcepetion {
194             Matcher varAssignmentMatcher = Config.VAR_ASSIGNMENT_PATTERN.matcher(currentLine);
195             // Match data members without values
```

```
196             // No scanning of local vars inside methods
197             if (varAssignmentMatcher.lookingAt()) {
198                 String varName = varAssignmentMatcher.group(1);
199                 String varValue = varAssignmentMatcher.group(2);
200                 try {
201                     method.setVariable(varName, varValue);
202                 } catch (InvalidVarNameExcepetion e) {
203                     // Try update a dataMember in case the var does not exist locally
204                     mainMethod.setVariable(varName, varValue);
205                 }
206                 return true;
207             }
208             return false;
209         }
210 
211         /**
212          * Match a line for a variable initialization without value assignment
213          * If matches - validates and adds the variable to the map
214          * @param currentLine the line to be matched
215          * @param method the method the line is in
216          * @return true if valid, false otherwise
217          * @throws DuplicateKeyException
218          * @throws InvalidVarValueException
219          * @throws InvalidReturnTypeException
220          * @throws FinalAssignmentException
221          * @throws InvalidVarNameExcepetion
222          */
223 
224         private static boolean parseVarNoInit(String currentLine,Method method) throws DuplicateKeyException,
225         InvalidVarValueException, InvalidReturnTypeException, FinalAssignmentException,
226         InvalidVarNameExcepetion {
227             Matcher varNoInitMatcher = Config.VAR_NO_INIT_PATTERN.matcher(currentLine);
228             // Match data members without values
229             // No scanning of local vars inside methods
230             if (varNoInitMatcher.lookingAt()) {
231                 String varName = varNoInitMatcher.group(2);
232                 String varType = varNoInitMatcher.group(1);
233                 validateVariableName(varName);
234                 method.createVariable(varName, varType, false);
235                 return true;
236             }
237             return false;
238         }
239 
240 
241 
242         /**
243          * Match a line for a final variable initialization with value assignment
244          * If matches - validates and adds the variable to the map
245          * @param currentLine the line to be matched
246          * @param method the method the line is in
247          * @return true if valid, false otherwise
248          * @throws InvalidArgumentException
249          * @throws InvalidVarNameExcepetion
250          * @throws DuplicateKeyException
251          * @throws InvalidReturnTypeException
252          * @throws FinalAssignmentException
253          * @throws InvalidVarValueException
254          * @throws NullReferenceException
255          */
256 
257         private static boolean parseVarFinalInit(String currentLine,Method method) throws InvalidArgumentException,
258         InvalidVarNameExcepetion, DuplicateKeyException, InvalidReturnTypeException,
259         FinalAssignmentException, InvalidVarValueException, NullReferenceException {
260             Matcher varInitFinalMatcher = Config.VAR_INIT_FINAL_PATTERN.matcher(currentLine);
261             if (varInitFinalMatcher.lookingAt()) {
262                 //Match the init parameters from the line
263                 String varType = varInitFinalMatcher.group(2);
```

```
            String varName = varInitFinalMatcher.group(3);
            String value = varInitFinalMatcher.group(4);
            boolean isFinal = Variable.parseFinal(varInitFinalMatcher.group(1));
            validateVariableName(varInitFinalMatcher.group(3));

            try {
                method.createVariable(varName, varType, value, isFinal);
            } catch (DuplicateKeyException e) {
                method.createVariable(varName, varType, mainMethod.getVariableValue(value), isFinal);
            }
            return true;
        }
        return false;
    }

    /**
     *
     * Match a line for a variable initialization with value assignment
     * If matches - validates and adds the variable to the map
     * @param currentLine the line to be matched
     * @param method the method the line is in
     * @return true if valid, false otherwise
     * @throws InvalidVarNameExcepetion
     * @throws DuplicateKeyException
     * @throws InvalidReturnTypeException
     * @throws FinalAssignmentException
     * @throws InvalidVarValueException
     * @throws NullReferenceException
     */

    private static boolean parseVarInitNotFinal(String currentLine,Method method) throws InvalidVarNameExcepetion,
    DuplicateKeyException, InvalidReturnTypeException, FinalAssignmentException,
    InvalidVarValueException, NullReferenceException {
        Matcher varInitNotFinalMatcher = Config.VAR_INIT_NOT_FINAL_PATTERN.matcher(currentLine);
        if (varInitNotFinalMatcher.lookingAt()) {
            String varType = varInitNotFinalMatcher.group(1);
            String varName = varInitNotFinalMatcher.group(2);
            String value = varInitNotFinalMatcher.group(3);
            validateVariableName(varName);
            try {
                method.createVariable(varName, varType, value, false);
            } catch (InvalidVarNameExcepetion e) {
                method.createVariable(varName, varType, mainMethod.getVariableValue(value), false);
            }
            return true;
        }
        return false;
    }

    /**
     * Match a line for a method call with variable initialization
     * Handles all cases such as final/not final variables and a call to an assigned variable
     * If matches - validates and adds the variable to the map
     * @param currentLine the line to be matched
     * @param method the method the line is in
     * @return true if valid, false otherwise
     * @throws InvalidVarNameExcepetion
     * @throws DuplicateKeyException
     * @throws InvalidVarValueException
     * @throws InvalidReturnTypeException
     * @throws FinalAssignmentException
     * @throws InvalidArgumentException
     * @throws InvalidMethodNameException
     * @throws UnknownLineException
     */

    private static boolean parseVarMethodAssignment(String currentLine,Method method)
    throws InvalidVarNameExcepetion, DuplicateKeyException,
```

```
332     InvalidVarValueException, InvalidReturnTypeException, FinalAssignmentException,
333     InvalidArgumentException, InvalidMethodNameException, UnknownLineException {
334         Matcher varFinalWithValueMatcher = Pattern.compile(
335                 Config.VAR_WITN_INIT_FINAL + Config.VALUES_METHOD_ASSIGNMENT).matcher(currentLine);
336         Matcher varNotFinalWithValueMatcher = Pattern.compile(
337                 Config.VAR_WITH_INIT_NOT_FINAL + Config.VALUES_METHOD_ASSIGNMENT).matcher(currentLine);
338         Matcher varAssignmentMatcher = Pattern.compile(
339                 Config.VAR_ASSIGNMENT + Config.VALUES_METHOD_ASSIGNMENT).matcher(currentLine);
340         if (varAssignmentMatcher.lookingAt()) {
341             try {
342                 validateCondition(method.getVariableReturnType(varAssignmentMatcher.group(1)),
343                         varAssignmentMatcher.group(2), method);
344                 method.setVariable(varAssignmentMatcher.group(1), null);
345                 return true;
346             } catch (InvalidVarNameExcepetion e) {
347                 validateCondition(method.getVariableReturnType(varAssignmentMatcher.group(1)),
348                         varAssignmentMatcher.group(2), mainMethod);
349                 mainMethod.setVariable(varAssignmentMatcher.group(1), null);
350             }
351         }
352         else if (varNotFinalWithValueMatcher.lookingAt()) {
353             validateVariableName(varNotFinalWithValueMatcher.group(2));
354             method.createVariable(varNotFinalWithValueMatcher.group(2),
355                     varNotFinalWithValueMatcher.group(1), false);
356             return (validateCondition(
357                     method.getVariableReturnType(varNotFinalWithValueMatcher.group(2)),
358                     varNotFinalWithValueMatcher.group(3), method));
359         }
360         if (varFinalWithValueMatcher.lookingAt()) {
361             validateVariableName(varFinalWithValueMatcher.group(3));
362             method.createVariable(varFinalWithValueMatcher.group(2),
363                     varFinalWithValueMatcher.group(3),
364                     Variable.parseFinal(varFinalWithValueMatcher.group(1)));
365             return (validateCondition(
366                     method.getVariableReturnType(varFinalWithValueMatcher.group(3)),
367                     varFinalWithValueMatcher.group(4), method));
368         }
369         return false;
370     }
371
372
373     /**
374      * Matches a method deceleration line,
375      * Add the method to the method map including its' input variables
376      * @return
377      * @throws InvalidMethodNameException
378      * @throws InvalidArgumentException
379      * @throws DuplicateKeyException
380      * @throws InvalidVarValueException
381      * @throws InvalidReturnTypeException
382      * @throws FinalAssignmentException
383      * @throws FileScanException
384      */
385
386     private static boolean parseMethodDecleration() throws InvalidMethodNameException,
387     InvalidArgumentException, DuplicateKeyException, InvalidVarValueException,
388     InvalidReturnTypeException, FinalAssignmentException, FileScanException {
389         Matcher methodMatcher = Config.METHOD_DECLERATION_PATTERN.matcher(codeLine);
390         if (methodMatcher.lookingAt()) {
391             String methodName = methodMatcher.group(2);
392             String returnType = methodMatcher.group(1);
393             String methodParams = methodMatcher.group(3);
394             validateMethodName(methodName);
395             methodMap.addMethod(methodName, returnType, methodParams);
396             codeScan.moveToLastMethodLine();
397             return true;
398         }
399         return false;
```

```
400         }
401
402         /**
403          * Matches a method deceleration line,
404          * Skip the current methods code and moves the scanner to the last line of the method
405          * @return true
406          * @throws FileScanException
407          */
408
409         private static boolean skipMethod() throws FileScanException {
410             Matcher methodMatcher = Config.METHOD_DECLERATION_PATTERN.matcher(codeLine);
411             if (methodMatcher.lookingAt()) {
412                 codeScan.moveToLastMethodLine();
413                 return true;
414             }
415             return false;
416         }
417
418         /**
419          * Scans the file for global data members lines
420          * and adds them to the map of variable map of the global method
421          * @throws InvalidArgumentException
422          * @throws FileScanException
423          * @throws UnknownLineException
424          * @throws DuplicateKeyException
425          * @throws InvalidVarValueException
426          * @throws InvalidReturnTypeException
427          * @throws FinalAssignmentException
428          * @throws InvalidVarNameExcepetion
429          * @throws InvalidMethodNameException
430          * @throws NullReferenceException
431          */
432
433         private static void preScanVars() throws InvalidArgumentException, FileScanException,
434         UnknownLineException, DuplicateKeyException, InvalidVarValueException,
435         InvalidReturnTypeException, FinalAssignmentException, InvalidVarNameExcepetion,
436         InvalidMethodNameException, NullReferenceException {
437             while (codeScan.hasNextLine()) {
438                 codeScan.nextLine();
439                 codeLine = codeScan.getLine();
440                 //Match only data members declarations
441                 if (parseVarNoInit(codeLine,mainMethod)) ;
442                 else if (parseVarInitNotFinal(codeLine,mainMethod)) ;
443                 else if (parseVarFinalInit(codeLine,mainMethod)) ;
444                 else if (parseVarMethodAssignment(codeLine, mainMethod)) ;
445                 else if (skipMethod()) ;
446                 else if (parseEmptyLine()) ;
447                 else
448                     throw new UnknownLineException(codeLine);
449             }
450         }
451
452         /**
453          * Scans the file for methods declerations
454          * and add the methods to the method map
455          * @throws FileScanException
456          * @throws InvalidMethodNameException
457          * @throws InvalidArgumentException
458          * @throws DuplicateKeyException
459          * @throws InvalidVarValueException
460          * @throws InvalidReturnTypeException
461          * @throws FinalAssignmentException
462          * @throws UnknownLineException
463          */
464         private static void preScanMethods() throws FileScanException,
465         InvalidMethodNameException, InvalidArgumentException, DuplicateKeyException, InvalidVarValueException
466         , InvalidReturnTypeException, FinalAssignmentException, UnknownLineException {
467             while (codeScan.hasNextLine()) {
```

```
468             codeScan.nextLine();
469             codeLine = codeScan.getLine();
470             //Match only method declaration lines
471             if (parseMethodDecleration());
472         }
473     }
474 
475     /**
476      * Parses an empty line in the file
477      * @return true if the line is matched, false otherwise
478      */
479 
480     private static boolean parseEmptyLine() {
481         Matcher emptyLineMatcher = Config.EMPTY_LINE_PATTERN.matcher(codeLine);
482         if (emptyLineMatcher.lookingAt()) return true;
483         else if (codeLine.length() ==0) return true;
484         return false;
485     }
486 
487 
488     /**
489      * Checks that the given string is a valid method name
490      *
491      * @param methodName the string which is the method name
492      * @throws InvalidMethodNameException in case that the method has an invalid name
493      */
494 
495     private static void validateMethodName(String methodName) throws InvalidMethodNameException {
496         if (!Config.METHOD_NAME_PATTERN.matcher(methodName).lookingAt()) {
497             throw new InvalidMethodNameException(methodName);
498         }
499     }
500 
501     /**
502      * Checks that the given string is a valid variable name
503      *
504      * @param variableName the string which is the variable name
505      * @throws InvalidVarNameExcepetion in case that the method has an invalid name
506      */
507 
508     private static void validateVariableName(String variableName)
509     throws InvalidVarNameExcepetion {
510         if (!Config.VARIABLE_NAME_PATTERN.matcher(variableName).lookingAt()) {
511             throw new InvalidVarNameExcepetion(variableName);
512         }
513     }
514     /**
515      * Match an if/while statement line, if matched validates the the inner condition is boolean
516      * @param currentLine the line to be matched
517      * @param method the method the line is in
518      * @return
519      * @throws InvalidVarNameExcepetion
520      * @throws InvalidArgumentException
521      * @throws InvalidVarValueException
522      * @throws InvalidMethodNameException
523      * @throws UnknownLineException
524      * @throws InvalidReturnTypeException
525      */
526 
527     private static boolean parseIfWhileStatement(String currentLine,Method method) throws InvalidVarNameExcepetion,
528     InvalidArgumentException, InvalidVarValueException,
529     InvalidMethodNameException, UnknownLineException, InvalidReturnTypeException {
530         Matcher ifWhileMatcher = Config.IF_WHILE_PATTERN.matcher(currentLine);
531         if (ifWhileMatcher.lookingAt()) {
532             String condition = ifWhileMatcher.group(1);
533             validateCondition(RETURN_TYPES.BOOLEAN, condition, method);
534             return true;
535         }
```

```
536             return false;
537         }
538 
539         /**
540          * Match a return statement, if matched validates that return value
541          * matches the one in the method definition
542          * @param currentLine the line to be matched
543          * @param method the method the line is in
544          * @return
545          * @throws InvalidReturnTypeException
546          * @throws InvalidVarNameExcepetion
547          * @throws InvalidArgumentException
548          * @throws InvalidVarValueException
549          * @throws InvalidMethodNameException
550          * @throws UnknownLineException
551          */
552 
553         private static boolean parseReturnStatement(String currentLine,Method method) throws InvalidReturnTypeException,
554         InvalidVarNameExcepetion, InvalidArgumentException, InvalidVarValueException,
555         InvalidMethodNameException, UnknownLineException {
556             Matcher returnMatcher = Config.RETURN_PATTERN.matcher(currentLine);
557             if (returnMatcher.lookingAt()) {
558                 if (returnMatcher.group(1).length() == 0) {
559                     //Return line should be empty only if the method is of type VOID
560                     if (method.getReturnType().equals(RETURN_TYPES.VOID)) return true;
561                     else if (!method.getReturnType().equals(RETURN_TYPES.VOID))
562                         throw new InvalidReturnTypeException(currentLine);
563                 }
564                 else if (method.getReturnType().equals(RETURN_TYPES.VOID)) {
565                     throw new InvalidReturnTypeException(currentLine);
566                 }
567                 else
568                     return (validateCondition(method.getReturnType(), returnMatcher.group(1), method));
569             }
570             return false;
571         }
572 
573         /**
574          * Checks that the given string is a valid condition
575          * Recursively checks the condition and validate the inner conditions
576          *
577          * @param conditionString the string which is the condition
578          * @throws InvalidVarNameExcepetion
579          * @throws InvalidArgumentException
580          * @throws InvalidVarValueException
581          * @throws InvalidMethodNameException
582          * @throws UnknownLineException
583          * @throws InvalidReturnTypeException
584          */
585         private static boolean validateCondition(RETURN_TYPES expectedType, String conditionString,
586                 Method method) throws InvalidVarNameExcepetion, InvalidArgumentException,
587                 InvalidVarValueException, InvalidMethodNameException, UnknownLineException, InvalidReturnTypeException {
588             //Remove outer spaces in the condition
589             conditionString = conditionString.trim();
590             Matcher bracketsMatcher = Config.BRACKETS_PATTERN.matcher(conditionString);
591             if (bracketsMatcher.lookingAt() && Method.splitCondition(conditionString).length == 1) {
592                 //Gets the method name and input variables using the regex capturing groups
593                 String methodName = bracketsMatcher.group(1);
594                 String methodParams = bracketsMatcher.group(2).trim();
595                 bracketsMatcher = bracketsMatcher.pattern().matcher(methodParams);
596                 /*
597                  * This block recursively gets the inner condition of the given condition and the condition at every step
598                  * It also validates that the number and type of the given variables matches the methods deceleration
599                  */
600                 if (Method.splitCondition(methodParams).length > methodMap.getMethod(methodName)
601                         .getInputParametersCount()) {
602                     if (methodMap.getMethod(methodName).returnsBool()) {
603                         if (Method.splitCondition(methodParams)[0].isEmpty()) return true;
```

```
                }
                else throw new InvalidArgumentException(codeLine);
            }
            //This is for a case of call where there is a single input parameters in the condition
            else if (Method.splitCondition(methodParams).length == 1 && method.getInputParametersCount() > 0) {
                try {
                    return validateCondition(method.getVariableReturnType(method.getKeyList().get(0)), methodParams, method)
                } catch (InvalidArgumentException e) {
                    return validateCondition(methodMap.getReturnType(methodName),
                            methodParams, methodMap.getMethod(methodName));
                }
            }
            else if (!bracketsMatcher.find()) {
                if (!methodMap.getReturnType(methodName).equals(expectedType))
                    throw new InvalidReturnTypeException(
                            conditionString,expectedType.toString(),methodMap.getReturnType(methodName).toString());
            }
            return validateCondition(methodMap.getReturnType(methodName), methodParams,
                    methodMap.getMethod(methodName));
        } else if (Method.splitCondition(conditionString).length > 1) {
            //Validate every parameter given the condition with the methods input parameters
            for (int i = 0; i < Method.splitCondition(conditionString).length; i++) {
                validateCondition(method.getVariableReturnType(method.getKeyList().get(i)),
                        Method.splitCondition(conditionString)[i].trim(), method);
            }
            //In case no exception is thrown while validating - returns true
            return true;
        } else {
            if (conditionString.isEmpty() && method.getReturnType().equals(expectedType)) return true;
            try {
                /*
                 * This block handles the logic of finding out what does the value contain:
                 * 1. Check if the value is an actual value like "5", if not:
                 * 2. Check if a variable with that name exists and initialized in the current scope, if not:
                 * 3. Check if a variable with that name exists and initialized as a data members, if not:
                 * 4. Throw an exception because such value is undefined
                 * in all first three cases - match the given type to the expected one
                 */
                if (Variable.parseVariable(expectedType, conditionString) == null)
                    throw new InvalidVarValueException(conditionString);
            } catch (InvalidVarValueException e) {
                try {
                    if (Variable.parseVariable(expectedType, method.getVariableValue(conditionString)) == null)
                        throw new InvalidVarValueException(expectedType + conditionString);
                } catch (InvalidVarNameExcepetion e2) {
                    try {
                        if (Variable.parseVariable(expectedType,
                                mainMethod.getVariableValue(conditionString)) == null)
                            throw new InvalidVarValueException(conditionString);
                    } catch (InvalidVarNameExcepetion e3) {
                        throw new InvalidArgumentException(conditionString);
                    }
                }
            }
            //In case no exception is thrown while validating - returns true
            return true;
        }
    }

    /**
     * Closes the code scanner
     */
    public static void closeScanner() {
        codeScan.closeFile();
    }
}
```

23 oop/ex2/scanner/CodeScanner.java

```java
package oop.ex2.scanner;

import java.io.File;
import java.io.FileReader;
import java.util.Scanner;
import java.util.regex.Pattern;

import oop.ex2.exceptions.CodeFileNotFoundException;
import oop.ex2.exceptions.FileScanException;

/**
 * A class that wraps a Scanner and scans a s-java file
 *
 * @author kobi_atiya 301143244 kobi atiya
 *
 */

public class CodeScanner {
    private Scanner codeFileScanner;
    private String codeLine;
    private String lastCodeLine;

    private CodeScanner(String fileString) {
        codeFileScanner = new Scanner(fileString);
        codeLine = "";
        lastCodeLine = "";
    }

    /**
     * A constructor for the CodeScanner class
     *
     * @param codeFile the s-java code file
     * @throws CodeFileNotFoundException
     */
    public CodeScanner(File codeFile) throws CodeFileNotFoundException {
        try {
            this.codeFileScanner = new Scanner(new FileReader(codeFile));
            codeLine = "";
            lastCodeLine = "";
        } catch (java.io.FileNotFoundException e) {
            throw new CodeFileNotFoundException(codeFile.getPath());
        }
    }

    /**
     * Extracts the block inside the {} block
     *
     * @return a scanner containing the lines inside the {} block
     * @throws FileScanException in case of an error when scanning the code file
     */
    public CodeScanner extractBlock() throws FileScanException {
        String result = "";
        int counter = 1;

        /*
         * This section checks for occurrence of a parentheses start and end,
         * and returns a new code scanner that contains the lines inside this block
         */

```

```
60              nextLine();
61              do {
62                  if (codeLine.contains(Config.PARENTHESES_START))
63                      counter++;
64                  if (codeLine.contains(Config.PARENTHESES_END))
65                      counter--;
66                  if (counter > 0) {
67                      result += codeLine + "\n";
68                      lastCodeLine = codeLine;
69                      nextLine();
70                  }
71
72              } while (counter > 0 && hasNextLine());
73
74              return (new CodeScanner(result));
75          }
76
77          /**
78           * Tells if the scanner has a next line
79           *
80           * @return true if the scanner has a next line, false otherwise
81           */
82          public boolean hasNextLine() {
83              return codeFileScanner.hasNextLine();
84          }
85
86          /**
87           * Gets the scanner's current line
88           *
89           * @return the current line
90           */
91          public String getLine() {
92              //Removes comments inside a single line
93              return (codeLine.replaceAll(Config.INLINE_COMMENT, " "));
94          }
95
96          /**
97           * Gets the last line of the scanner
98           * Used in getting the last line after block extraction
99           * @return the last line of this scanner
100          */
101         public String getLastLine() {
102             return (lastCodeLine);
103         }
104
105         /**
106          * Moves the scanner to the next line, Skips comments and empty lines
107          *
108          * @throws FileScanException in case of an error when scanning the code file
109          */
110         public void nextLine() throws FileScanException {
111             if (codeFileScanner.hasNextLine())
112                 codeLine = codeFileScanner.nextLine();
113
114             /*
115              * The next line returned will be the next line that is not a comment or an empty line,
116              * in that way all comments in the code file are ignored
117              */
118
119             while ((codeLine.startsWith((Config.SINGLE_COMMENT)) || codeLine.matches(Config.EMPTY_LINE) || codeLine
120                     .length() == 0) && hasNextLine()) {
121                 codeLine = codeFileScanner.nextLine();
122             }
123
124             if (codeLine.startsWith(Config.COMMENT_SECTION_START)) {
125                 if (codeLine.contains(Config.COMMENT_SECTION_END)) {
126                     codeLine = codeFileScanner.nextLine();
127                 }
```

```
128                 else
129                 while (!codeLine.contains(Config.COMMENT_SECTION_END)) {
130                     if (!hasNextLine())
131                         throw new FileScanException(codeLine);
132                     codeLine = codeFileScanner.nextLine();
133                 }
134                 codeLine = codeFileScanner.nextLine();
135             }
136 
137         }
138 
139         /**
140          * Moves the scanner to the next method in the code file
141          * @throws FileScanException in case of an error when scanning the code file
142          */
143         public void nextMethodStartLine() throws FileScanException {
144             Pattern methodPattern = Pattern.compile(Config.METHOD_DECLERATION);
145             while (!methodPattern.matcher(codeLine).matches() && hasNextLine()) {
146                 lastCodeLine = codeLine;
147                 nextLine();
148             }
149         }
150 
151         /**
152          * Moves the scanner to the last line of the current method,
153          * assumes that the current line is a method declaration line
154          * @throws FileScanException in case of an error when scanning the code file
155          */
156         public void moveToLastMethodLine() throws FileScanException {
157             int parenthesesCounter = 0;
158             do {
159                 if (codeLine.contains(Config.PARENTHESES_START))
160                     parenthesesCounter++;
161                 if (codeLine.contains(Config.PARENTHESES_END))
162                     parenthesesCounter--;
163                 if (parenthesesCounter > 0) {
164                     lastCodeLine = codeLine;
165                     nextLine();
166                 }
167             } while (parenthesesCounter > 0 && hasNextLine());
168 
169         }
170 
171         /**
172          * Close the scanner
173          */
174         public void closeFile() {
175             codeFileScanner.close();
176         }
177 
178     }
```

24 oop/ex2/scanner/Config.java

```java
package oop.ex2.scanner;

import java.util.regex.Pattern;

/**
 * A configuration class containing all different constants used
 *
 * @author kobi_atiya 301143244 kobi atiya
 */
public class Config {

    // Comments and empty line constants - ignored
    public static final String SINGLE_COMMENT = "//";
    public static final String COMMENT_SECTION_START = "/*";
    public static final String COMMENT_SECTION_END = "*/";
    public static final String EMPTY_LINE = "(?m)^\\s*$[\n\r]{1,}";

    // Block start and end
    public static final String PARENTHESES_START = "{";
    public static final String PARENTHESES_END = "}";

    // Method declaration
    public static final String METHOD_DECLERATION = "^\\s*(\\w*)\\s*(\\w*)\\s*\\((.*)\\)\\s*\\{";
    public static final String METHOD_NAME = "^(?!_)[A-Za-z_]\\w*";

    // Variable declaration regex
    public static final String VARIABLE_VALUE = "([-{0,1}\\w.*]*|\\\".*\\\"|'.*');\\s*";
    public static final String VAR_WITH_INIT_NOT_FINAL = "^\\s*(\\w*)\\s*(\\w*)\\s*=\\s*";
    public static final String VAR_WITN_INIT_FINAL = "^\\s*(\\w*)\\s*(\\w*)\\s*(\\w*)\\s*=\\s*";
    public static final String VAR_WITHOUT_INIT = "^\\s*(\\w*)\\s*(\\w*)\\s*;";

    // Variable assignment regex
    public static final String VAR_ASSIGNMENT = "^\\s*(\\w*)\\s*=\\s*";
    public static final String VALUES_METHOD_ASSIGNMENT = "(.*);";
    public static final String METHOD_CALL = "^\\s*(\\w*)\\((.*)\\);\\s*";

    // Return lines
    public static final String RETURN_LINE = "^\\s*return\\s*(.*);";

    // An if/while line
    public static final String IF_WHILE_STATEMENT = "^\\s*(?:if|while)\\s*\\((.*)\\)\\s*\\{";

    public static final String VAR_NAME = "^[A-Za-z]\\w*|^[_]\\w+";
    public static final String BRACKETS = "(\\w*)\\s*\\((.*)\\)";
    public static final String INLINE_COMMENT = "/\\*.*\\*/";
    public static final String SPACES_CHARS = "\\s+";

    //Default values for variables
    public static final String STRING_DEFAULT_VALUE = "\"\"";
    public static final String BOOLEAN_DEFAULT_VALUE = "0";
    public static final String CHAR_DEFAULT_VALUE = "''";

    // Regex patterns
    public static Pattern VAR_NO_INIT_PATTERN = Pattern.compile(Config.VAR_WITHOUT_INIT);
    public static Pattern VAR_INIT_NOT_FINAL_PATTERN = Pattern.compile(Config.VAR_WITH_INIT_NOT_FINAL+Config.VARIABLE_VALUE);
    public static Pattern VAR_INIT_FINAL_PATTERN = Pattern.compile(Config.VAR_WITN_INIT_FINAL+Config.VARIABLE_VALUE);
    public static Pattern METHOD_DECLERATION_PATTERN = Pattern.compile(Config.METHOD_DECLERATION);
    public static Pattern METHOD_NAME_PATTERN = Pattern.compile(Config.METHOD_NAME);
    public static Pattern VARIABLE_NAME_PATTERN = Pattern.compile(Config.VAR_NAME);
```

```
60      public static Pattern VAR_ASSIGNMENT_PATTERN = Pattern.compile(VAR_ASSIGNMENT+Config.VARIABLE_VALUE);
61      public static Pattern IF_WHILE_PATTERN = Pattern.compile(IF_WHILE_STATEMENT);
62      public static Pattern BRACKETS_PATTERN = Pattern.compile(BRACKETS);
63      public static Pattern RETURN_PATTERN = Pattern.compile(RETURN_LINE);
64      public static Pattern EMPTY_LINE_PATTERN = Pattern.compile(EMPTY_LINE);
65  }
```