Contents

1 Basic Test Results

```
Logins: liat_gin



compiling with
    javac -cp .:/cs/course/2013/oop/lib/junit4.jar  *.java oop/ex7/main/*.java


tests output :
        Perfect!
```

2 README

```
liat_gin
oranpa

===========
EX7 README:
===========

============================
Description of the classes:
============================



oop.ex7.blocks:
===============

1) Block: an abstract class that represents any kind of block in the program.
   Any block starts with "{" and ends with: "}" (not includes arrays).
   Block can be: The whole program, method and if or while.
   Contains the abstract method "checkLineAllowed" which decides whether
   the given line in the block is legal line or not. Therefore any class
   that extends from block has to implement
   this method.



2) ConditionFactory: class that creates a new condition block :
   If block or while block.

3) Conditions: the father class of If and While.Represents block of the
   type condition. Extends from InnerBlock Class.

4) GlobalBlock: represents the global block which means the block of
   the whole file (if we think about it as a class so globalBlock is the all class).
   Extends from Block.

5) If: represents a block of If. Extends from Condition.

6) InnerBlock: represents any block that is not the global block which includes
   methods and condition blocks (if and while). Extends from Block.

7) Method: represents a method block. Extends from InnerBlock.

8) While: represents a while block. Extends from Condition.
    EXCEPTIONS:
    BlockException.java- thrown when there is a problem in the block.
    IfBlockException.java- thrown when there is a problem with the if block.
    MethodException.java-thrown when there is a problem with the method.
    NoSuchBlock-thrown if the method doesnt exist.
    NoSuchLine-thrown when the line is not exist.
    WrongNumberOfException-throw when there is problem with
    the number of bracket.



oop.ex7.main:
=============

```

9) IllegalCodeException: exception that being called any time the code in
   sjavac is not a legal code. prints an informative massage about the specific
   error that occurred.

10) Sjavac: the "main" class that contains the main method. In that class the first
    reading from the file is happening and this is the first pass on the file which
    "cleans" the file from comments, ";" literals and checks brackets validity.
    After the first read sends the file to pars process (which parse the methods
    and the all other blocks in the file).

11) Tools: this class contains useful methods that being used several times
    in the program.

12) Variable: represents a Variable in the program. It can be a primitive
    variable or not primitive variable (array). Contains all the essential
    data on the variable.
    EXCEPTIONS:
    DuplicateDeclaration.java-exception thrown when duplicate declaration
       IllegalCodeException.java- thrown whenever there is a problem in the file
       InvalidValue.java-thrown if we put illegal value.
       NoInitilizedVariableException.java-thrown if the var is not Initilized.
       VariableNotExist.java-thrown if the var is not exist.

oop.ex7.regex:
==============

13) RegexBox: this class contains all the regular expressions that we used
    in the program.

oop.ex7.types:
==============
14) Type: represents the type of the variable in the program.

15) TypeFactory: creates a new Variable according to the given type.

16) IntegerVariable: represents an integer type of value. Extends from Type.

17) DoubleVariable: represents an double type of value. Extends from Type.

18) BooleanVariable: represents an boolean type of value. Extends from Type.

19) StringVariable: represents an string type of value. Extends from Type.

20) CharVariable: represents an char type of value. Extends from Type.

21) WrongValueException: exception that being called when the value is not a
    legal value. Extends from IllegalCodeException.
 EXCEPTIONS:
 TypeException.java-thrown if there is a problem with the type.
 NoSuchType-thrown if we put invalid type to the var.

====================
General desciption:
====================

1. The design of ex7 : our code divided to 4 modules.

The first module is the main.
It's the module that suppose to connect between the other modules. Its class
contains the sjavac which go over the file, first it read it to list of lines
and then parse the file by creating a global block which is the file
itself and divide the file into methods and global variables.
The other classes are class called Tools which contain some methods that been used
for the entire program.

There are methods which their role is to remove spaces in the file, check if the line

suppose to contain point in the end of it and check if the program has
valid number of brackets. It contains also a class that represent the structure of
vriable.Variable contains the variable name, type, value, isPrimitive member
(which tell us if it is an array or not) and the block which contains the certain
variable.

Other classes are classes of exceptions an exception which called
illegalCodeException which is the the father of all exceptions which
suppose to print 1 in the program meaning there is an error in the code.

The second module is Block which contains the Block class. This class is the
one who responsible to parse blocks in the program, a block in our definition
is every thing between open bracket to close one (not including arrays of course)
including the file itself. The Block class is an abstract class and it is the
father block of global block and inner block.We chose to make Block class as abstract
because it represents block in general but not a specific one and therefore
it is not necessary to create an instance of it cause it too general.

The inner block represent blocks which are in the file and global represent
the entire file. Inner block which are method, if or while blocks.
The block class responsible to parse every block in the program and
every other class represent the structure of every block in the program.
The inner blocks the hierarchy of the block module is the block class represent
every block in the program and parse them his sons are global block and inner block
inner blocks sons are method and conditions.Class Method represents method
and condition represent condition block like if and while and the sons of
conditions are If and While classes. Other classes are exceptions which occur
in the block module.

The third module is regex module which contains the regex's which used
 in the program.

The fourth module is types. Type is module that suppose to manage the type
of the variable. It suppose to check if the type and the value of the variable
are match. The class type is the class which checks if the type of the
variable and it's value are match. This class is the father class of every type
of variable which includes: boolean, integer char, double and String.
It checks whether the assignment matches the type of the variable.

2. Regular expression: here is regex that checks the type. The regex called
"valid type" this regex checks if the type of the variable is legal. It checks if
it's int ,boolean, char, double or String and if it contains []
(if the type is an array or not). Other regex is regex that checks if the variable
name is legal or not, if it contain "_" in the beginning or not.

3. Error handling: in this department we constructed types of exception
for every mistake the class IllgalCodeException is the class that represents
the type 1 errors and through the program there are exceptions which inherit
from this class like exception which thrown when there is problem with creation
a method or problem with creation If block or there is a mismatch between the
type and the value of the variable the appropriate exception are been thrown and an
appropriate message which say in one line what causes the problem.

==========
Answering to questions:
==========

4. If we have to add new variables types to the program we will modify
our program as following: for any new type we will define a new class
called "NewTypeVariable" which extends the class Type and we will modify
the TypeFactory that it will be able to create new instances of the new
types variables.

5. If we will have to support if-else block it will be done as follows:
we will define a new class called "Else" which will extends from If.
The Else class will be considered as new kind of block. Therefore
the father block of it will be the relevant If block.

The local variables of else block will be the same of its father If block
because it can use the same variables that were defined in the If block
or to create new variables in it.
In addition we will update our regex box by creating new regex who finds
a line that starts with "else" expression and that will follow a little
changes in other regexes (because from now line that starts with "else"
will be a legal line).

-

3 oop/ex7/blocks/Block.java

```java
package oop.ex7.blocks;
import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import oop.ex7.main.DuplicateDeclaration;
import oop.ex7.main.IllegalCodeException;
import oop.ex7.main.InvalidDeclaration;
import oop.ex7.main.InvalidValue;
import oop.ex7.main.NotInitlizedVariableException;
import oop.ex7.main.Tools;
import oop.ex7.main.Variable;
import oop.ex7.main.VariableNotExistException;
import oop.ex7.regex.RegexBox;
import oop.ex7.types.ArrayNotExistException;
import oop.ex7.types.TypeException;
/**
 * Class Block
 *represents any block in the program inner block
 *(methods, and if and while block) and global block which
 *includes the whole program block.
 */
public abstract class Block {
    public static final String PARAMETER_COUNT_ERROR= "Invalid parameter count";
    public static final String ILLEGAL_METHOD_RETURN_TYPE= "Illegal method return type";
    public static final String METHOD_ERROR="Method not found";
    public static final String ARRAY_NOT_EXIST="Array does not exist";
    public static final String NOT_INTILIZED_VAR="The variable was not initialized";
    public static final String MISMATCH_TYPE_VALUE="the value doesnt match the target type";
    public static final String INVALID_VALUE="invalid value";
    public static final String ILLIGAL_EXPRESSION="illegal expression";
    public static final String NO_SUCH_LINE="no such line";
    public static final String WRONG_NUMBER_OF_BRACKET="Wrong number of brackets";
    public static final String INVALID_DEC ="Invalid declaration";
    public static final String DUPLICATE_VAR ="duplicate varaiable";
    public static final String INVALID_ASSIGN="Invalid assignment";
    public static final String NOT_EXIST_VAR="Varaible is not exist";
    public static final String ILLGAL_DEC_ASSIGN="Invalid declaration and assignment";
    public static final String INVALID_TYPE="Invalid target type";
    public static final String INVALID_OPERATOR_MINUS="Minus allowed only on numbers";
    public static final String COMMA = ",";
    public static final String EMPTY = "";
    public static final String BARCKET = "[]";
    public static final String INT = "int";
    public static final String BOOL = "boolean";
    public static final String DOUBLE = "double";
    private static final int NOT_FOUND = -1;
    protected String nameOfBlock;
    protected Block father;
    protected ArrayList<Variable> localVaribles = new ArrayList<Variable>();
    protected ArrayList <Method> methods = new ArrayList<Method>();
    protected ArrayList<String> content= new ArrayList<String>();

    /**
     * constructor.
     * @param nameOfBlock the name of the block.
     * @param father the father block.
     * @param content the lines that inside that block.
     */
```

```
60      public Block (String nameOfBlock, Block father, ArrayList<String> content) {
61          this.nameOfBlock = nameOfBlock;
62          this.father = father;
63          this.content = content;
64      }
65
66      /**
67       * @return an array list of the local variables.
68       */
69      public ArrayList<Variable> getLocalVariables() {
70          return this.localVaribles;
71      }
72
73      /**
74       * @return the block's father.
75       */
76      public Block getFather() {
77          return this.father;
78      }
79
80      /**
81       * @return an array list of the block's content.
82       */
83      public ArrayList<String> getContent(){
84          return this.content;
85      }
86
87      /**
88       *
89       * @param line a line to check its legality.
90       * @return true iff the given line is legal line.
91       * @throws IllegalCodeException
92       */
93      protected abstract boolean checkLineAllowed(String line)
94              throws IllegalCodeException;
95
96      /**
97       * @return an array list of the methods blocks.
98       */
99      public ArrayList<Method> getMethods() {
100         return methods;
101     }
102
103     /**
104      * sends a certain method to parse.
105      * @param returnedValueType the type value the method returners.
106      * @param father the father of the method.
107      * @param content lines that inside that method.
108      * @throws IllegalCodeException
109      */
110     public void parseMethod
111     (String returnedValueType, Block father, ArrayList<String> content)
112             throws IllegalCodeException {
113         parseMethod(returnedValueType, father, content, false);
114     }
115
116     /**
117      * parse a given method.
118      * @param returnedValueType the type value the method returners.
119      * @param father the father of the method.
120      * @param content lines that inside that method block.
121      * @param addMethods a boolean var who says if this method already exist in the list of the methods or not.
122      * @throws IllegalCodeException
123      */
124     public void parseMethod
125     (String returnedValueType, Block father, ArrayList<String> content, boolean addMethods)
126             throws IllegalCodeException {
127
```

```
128         if (Tools.checkBracketsValidity(content)) { //the number of brackets is valid
129
130             for(int i = 0; i < content.size(); i ++) {
131
132                 String line = content.get(i);
133
134                 line = Tools.whichKindOfLine(content.get(i));
135
136                 if (!checkLineAllowed(line)) {
137                     // Skip this line
138                     continue;
139                 }
140
141                 switch (line) {
142
143                 case Tools.COMMENT:
144                 case Tools.EMPTY_LINE:
145                     break;
146
147                 case Tools.METHOD_REGEX : {
148
149                     ArrayList<String> methodContent =
150                             Tools.findNewContent(content, content.get(i), i);
151                     Method method = new Method(content.get(i), this, methodContent);
152
153                     if (addMethods) {
154                         methods.add(method);
155                     }
156
157                     i = i + methodContent.size() + 1;
158
159                     break;
160                 }
161
162                 case Tools.DECLARATION_ON_VAR : {
163
164                     Pattern paternDeclareOnVar =
165                             Pattern.compile(RegexBox.DECLARATION_ON_VAR);
166                     Matcher matchDeclareOnVar =
167                             paternDeclareOnVar.matcher(content.get(i));
168
169                     if (!matchDeclareOnVar.matches()) {
170                         throw new InvalidDeclaration(INVALID_DEC);
171                     }
172
173                     Variable newVar =
174                             new Variable
175                             (this, matchDeclareOnVar.group(1)
176                                     ,matchDeclareOnVar.group(4));
177                     if(isNonGlobalVarExist(newVar.getName()) != null) {
178                         throw new DuplicateDeclaration(DUPLICATE_VAR);
179                     }
180                     this.localVaribles.add(newVar);
181                     break;
182                 }
183
184                 case Tools.ASSIGNING_IN_VAR : {
185                     Pattern paternAssignOnVar =
186                             Pattern.compile(RegexBox.ASSIGNING_ON_VAR);
187                     Matcher matchAssignOnVar =
188                             paternAssignOnVar.matcher(content.get(i));
189                     if(!matchAssignOnVar.matches()){
190                         throw new InvalidValue(INVALID_ASSIGN);
191                     }
192                     String nameOfVar = matchAssignOnVar.group(2);//splitedLine[0];
193                     String value = matchAssignOnVar.group(7);//splitedLine[1];
194                     String type = null;
195
```

```
196                     if (nameOfVar != null) {
197                         Variable targetVar = isVarExist(nameOfVar);
198 
199                         if (targetVar == null) {
200                             throw new VariableNotExistException(NOT_EXIST_VAR);
201                         }
202 
203                         type = targetVar.getType();
204 
205                         targetVar.setValue();
206                     }
207                     else {
208                         type = getArrayType(matchAssignOnVar.group(1));
209 
210                         if (type == null) {
211                             throw new InvalidValue(INVALID_ASSIGN);
212                         }
213                     }
214 
215                     validAssigningPossibilities(type, value);
216                     break;
217                 }
218 
219                 case Tools.DECLARATION_AND_ASSIGNINNG_IN_VAR : {
220                     String type = null;
221                     String value = null;
222                     Pattern paternDeclareAndAssignOnVar =
223                             Pattern.compile(RegexBox.DECLARATION_AND_ASSIGING_ON_VAR);
224                     Matcher matchDeclareAndAssignOnVar =
225                             paternDeclareAndAssignOnVar.matcher(content.get(i));
226                     if(!matchDeclareAndAssignOnVar.matches()) {
227                         throw new IllegalCodeException(ILLGAL_DEC_ASSIGN);
228                     }
229 
230                     type = Variable.normalizeType(matchDeclareAndAssignOnVar.group(1));
231                     value=matchDeclareAndAssignOnVar.group(5);
232 
233                     validAssigningPossibilities(type, value);
234 
235                     Variable newVar =
236                             new Variable(this,
237                                     matchDeclareAndAssignOnVar.group(1),
238                                     matchDeclareAndAssignOnVar.group(4), true);
239 
240                     if (isNonGlobalVarExist(newVar.getName()) != null) {
241                         throw new DuplicateDeclaration(DUPLICATE_VAR);
242                     }
243 
244                     this.localVaribles.add(newVar);
245                     break;
246                 }
247 
248                 case Tools.IF_OR_WHILE : {
249 
250                     ArrayList<String> newContent =
251                             Tools.findNewContent(content, content.get(i), i);
252                     Block newBlock =
253                             ConditionFactory.createCondition
254                             (content.get(i), newContent, this);
255                     newBlock.parseMethod
256                     (returnedValueType,newBlock, newContent, addMethods);
257                     i = i + newContent.size() + 1;// jumps over the new block
258                     break;
259                 }
260 
261                 case Tools.METHOD_CALL : {
262                     Pattern paternCall = Pattern.compile(RegexBox.METHOD_CALL);
263                     Matcher matchCall = paternCall.matcher(content.get(i));
```

```
                    if(matchCall.matches()){
                        checkMethodCall(null, matchCall.group(2), matchCall.group(3));
                    }
                }
                case Tools.RETURN_LINE : {
                    String returnedvar = null;
                    Pattern paternReturn = Pattern.compile(RegexBox.RETURN_LINE);
                    Matcher matchReturn = paternReturn.matcher(content.get(i));
                    if(matchReturn.matches()){
                        returnedvar = matchReturn.group(3);
                    }

                    if(returnedvar!=null) {
                        validAssigningPossibilities(returnedValueType, returnedvar);
                    }
                    break;
                }

                default:
                    throw new NoSuchLine(NO_SUCH_LINE);
                }
            }
        }
        else {
            throw new WrongNumberOfException(WRONG_NUMBER_OF_BRACKET);
        }
    }


    /**
     * checks whether a new assigning is possible according to the types of the new
     * assisgment value and the variable to assign to.
     * @param destination the type of the var to assign into him.
     * @param source the type of the new variable to assign into destination variable.
     * @return true iff the types of the destination and source are equals or if destination
     * is double and source is int. return false otherwise.
     */
    private boolean canAssign(String destination, String source) {
        destination = destination.trim();
        source = source.trim();

        if (destination.equals(source) ||
                (destination.equals(DOUBLE) && source.equals(INT))) {
            return true;
        }
        return false;
    }


    /**
     * @param nameOfVar variable to check.
     * @return the variable if its not a global variable
     * and null if its a global variable.
     */
    public  Variable isNonGlobalVarExist(String nameOfVar) {
        Variable var = isVarExist(nameOfVar);

        if (var != null && var.getBlock().getFather() == null) {
            var = null;
        }

        return var;
    }

    /**
     * search for a given name of a variable.
     * @param nameOfVar the name of the variable to search for.
     * @return the variable itself if it exist and null otherwise.
```

```
332         */
333        public  Variable isVarExist(String nameOfVar) {
334
335            Pattern paternArrayElement = Pattern.compile(RegexBox.ARRAY_ELEMENT);
336            Matcher matchArrayElement = paternArrayElement.matcher(nameOfVar);
337
338            if(matchArrayElement.find()){
339                nameOfVar=matchArrayElement.group(1);
340            }
341
342            Block block = this;
343
344            while (block != null) {
345                ArrayList<Variable> localVariables = block.getLocalVariables();
346
347                int indexOfVar = findIndexOf(nameOfVar, localVariables);
348
349                if(indexOfVar != NOT_FOUND) {
350                    return localVariables.get(indexOfVar);
351                }
352
353                block = block.getFather();
354            }
355            return null;
356        }
357
358        /**
359         *
360         * @param values a string of an array parameters.
361         * @return an array list of strings that contains the
362         * parameters of the array if exist.
363         */
364        public String[] splitArrayValue(String values) {
365            Pattern paternVarInsideBraket = Pattern.compile
366                    (RegexBox.VAR_INSIDE_BRAKET_ARRAY);
367            Matcher matchVarInsideBraket = paternVarInsideBraket.matcher(values);
368            if(matchVarInsideBraket.matches()){
369                return matchVarInsideBraket.group(1).split(COMMA, -1);
370            }
371            return null;
372
373        }
374        /**
375         * @param var a variable to search for.
376         * @param localVariables a list of the local variables that we have.
377         * @return the index of the given variable if it was found in the local
378         * variables list and -1 otherwise.
379         */
380        public  int findIndexOf(String nameOfVar, ArrayList<Variable> localVariable) {
381
382            for (int i = 0; i < localVariable.size(); i++ ) {
383
384                if(localVariable.get(i).getName().equals( nameOfVar) ) {
385                    return i;
386                }
387            }
388            return NOT_FOUND;
389        }
390
391        /**
392         * checks whether a certain assigning is legal and throws exception if
393         * its not  legal assignment.
394         * @param type the value type
395         * @param value the value
396         * @throws IllegalCodeException
397         */
398        public  void validAssigningPossibilities(String type, String value)
399                throws IllegalCodeException {
```

```
400
401         if (type.equals(INT) || type.equals(DOUBLE)) {
402             Pattern paternExp=Pattern.compile(RegexBox.DOUBLE_OPERATOR_EXPRESSION);
403             Matcher matchExp = paternExp.matcher(value);
404
405             if(!matchExp.matches()){
406                 throw new IllegalCodeException(ILLIGAL_EXPRESSION);
407             }
408
409             validSimpleAssigningPossibilities(type, matchExp.group(1));
410
411             if (matchExp.group(12) != null) {
412                 validSimpleAssigningPossibilities(type, matchExp.group(13));
413             }
414         }
415         else {
416             validSimpleAssigningPossibilities(type, value);
417         }
418     }
419
420     /**
421      *
422      * @param value the array to check.
423      * @return the type of a given array and throws an exception
424      * if the array is not exist or if its type isnt legal.
425      * @throws IllegalCodeException
426      */
427     public String getArrayType(String value) throws IllegalCodeException {
428         Pattern paternArrayElement = Pattern.compile(RegexBox.MINUS_ARRAY_ELEMENT);
429         Matcher matchArrayElement = paternArrayElement.matcher(value);
430         if(!matchArrayElement.matches()) {
431             return null;
432         }
433
434         String arrayName = matchArrayElement.group(2);
435         Variable arrayVar = isVarExist(arrayName);
436
437         if (arrayVar == null) {
438             throw new ArrayNotExistException(ARRAY_NOT_EXIST);
439         }
440
441         if (arrayVar.getIsPrimitive()) {
442             throw new TypeException(MISMATCH_TYPE_VALUE);
443         }
444
445         if (!arrayVar.hasValue()) {
446             throw new NotInitlizedVariableException(NOT_INTILIZED_VAR);
447         }
448
449         if (matchArrayElement.group(1) != null) {
450             throw new TypeException(INVALID_OPERATOR_MINUS);
451         }
452
453         String arrayType = arrayVar.getType();
454         String elementVarName = matchArrayElement.group(5);
455
456         if (elementVarName != null) {
457             Variable elementVar = isVarExist(elementVarName);
458
459             if (elementVar == null) {
460                 throw new VariableNotExistException(NOT_EXIST_VAR);
461             }
462
463             if (!elementVar.getIsPrimitive() ||
464                     !elementVar.getType().equals(INT)) {
465                 throw new InvalidValue(INVALID_VALUE);
466             }
467         }
```

```
468
469             return arrayType.replace(BARCKET, EMPTY);
470         }
471
472
473         /**
474          * checks whether a certain assigning is legal and throws
475          * exception if its not  legal assignment.
476          * @param type the value type
477          * @param value the value
478          * @throws IllegalCodeException
479          */
480         public void validSimpleAssigningPossibilities(String type,String value)
481                 throws IllegalCodeException {
482             value = value.trim();
483
484             Pattern paternBoolean = Pattern.compile(RegexBox.BOOLEAN_VALUE);
485             Matcher matchBoolean = paternBoolean.matcher(value);
486             if (matchBoolean.matches()) {
487                 if (!canAssign(type,BOOL)) {
488                     throw new InvalidValue(INVALID_TYPE);
489                 }
490                 return;
491             }
492
493             Pattern paternVarName = Pattern.compile(RegexBox.MINUS_VARIABLE_NAME);
494             Matcher matchVarName = paternVarName.matcher(value);
495             if(matchVarName.matches()){
496                 Variable var = isVarExist(matchVarName.group(2));
497
498                 if(var == null) {
499                     throw new VariableNotExistException(NOT_EXIST_VAR);
500                 }
501
502                 if(!canAssign(type, var.getType())){
503                     throw new TypeException(MISMATCH_TYPE_VALUE);
504                 }
505
506                 if (!var.hasValue()) {
507                     throw new NotInitlizedVariableException(NOT_INTILIZED_VAR);
508                 }
509
510                 if (matchVarName.group(1) != null &&
511                         !var.getType().equals(DOUBLE) &&
512                         !var.getType().equals(INT)) {
513                     throw new InvalidValue(INVALID_OPERATOR_MINUS);
514                 }
515
516                 return;
517             }
518
519             Pattern paternMethodName = Pattern.compile(RegexBox.MINUS_METHOD_CALL);
520             Matcher matchMethod = paternMethodName.matcher(value);
521             if(matchMethod.matches()){
522                 String methodName = matchMethod.group(3);
523                 checkMethodCall(type, methodName, matchMethod.group(4));
524
525                 if (matchMethod.group(1) != null &&
526                         !type.equals(DOUBLE) && !type.equals(INT)) {
527                     throw new InvalidValue(INVALID_OPERATOR_MINUS);
528                 }
529
530                 return;
531             }
532
533             String elementArrayType = getArrayType(value);
534
535             if (elementArrayType != null) {
```

```
            if (type.contains(BARCKET) ||
                    !canAssign(type, elementArrayType)) {
                throw new TypeException(INVALID_TYPE);
            }

            return;
        }

        String[] splitValues = splitArrayValue(value);
        if(splitValues==null){

            Variable.checkValidPrimitive(type, value);
        }
        else {
            Pattern paternArrayType = Pattern.compile(RegexBox.VALID_ARRAY_TYPE);
            Matcher matchArrayType = paternArrayType.matcher(type);
            if(!matchArrayType.matches()){
                throw new InvalidValue(MISMATCH_TYPE_VALUE);
            }

            if (splitValues.length == 1 && splitValues[0].trim().equals(EMPTY)) {
                // Return an empty array (OK)
                return;
            }

            String arrayType = matchArrayType.group(1);

            for(int j=0; j<splitValues.length; j++){
                splitValues[j] = splitValues[j].trim();

                Pattern paternElementBoolean = Pattern.compile(RegexBox.BOOLEAN_VALUE);
                Matcher matchElementBoolean = paternElementBoolean.matcher(splitValues[j]);
                if (matchElementBoolean.matches()) {
                    if (!canAssign(arrayType,BOOL)) {
                        throw new TypeException(INVALID_TYPE);
                    }
                    continue;
                }

                Pattern paternElementVar = Pattern.compile(RegexBox.MINUS_VARIABLE_NAME);
                Matcher matchElementVar = paternElementVar.matcher(splitValues[j]);
                if(matchElementVar.matches()){
                    String splitVarName = matchElementVar.group(2);
                    Variable splitVar= isVarExist(splitVarName);
                    if(splitVar!=null){
                        if(!canAssign(arrayType, splitVar.getType())){
                            throw new InvalidValue(MISMATCH_TYPE_VALUE);
                        }

                        if (!splitVar.hasValue()) {
                            throw new
                            NotInitlizedVariableException(NOT_INTILIZED_VAR);
                        }

                        if (matchElementVar.group(1) != null &&
                                !splitVar.getType().equals(DOUBLE) &&
                                !splitVar.getType().equals(INT)) {
                            throw new InvalidValue(INVALID_OPERATOR_MINUS);
                        }
                    }
                    else{
                        throw new VariableNotExistException(NOT_EXIST_VAR);
                    }
                }
                else
                {
                    Variable.checkValidPrimitive(arrayType, splitValues[j]);
                }
```

```
604                 }
605             }
606         }
607
608
609
610         /**
611          * checks whether a call to method is legal by check if this
612          * method is already exist and if it exists
613          * checks its return type and its parameters.
614          * @param type the return type of the method.
615          * @param name the name of the method.
616          * @param parameters the parameters  of the method
617          * @throws IllegalCodeException
618          */
619         protected void checkMethodCall(String type, String name, String parameters)
620                 throws IllegalCodeException {
621
622             for(Method method : getGlobalMethods()){
623                 if (!method.getMethodName().equals(name)) {
624                     continue;
625                 }
626
627                 if(type != null && !canAssign(type, method.getReturnedValue())) {
628                     throw new TypeException( ILLEGAL_METHOD_RETURN_TYPE);
629                 }
630
631                 String[] paramterTypes = method.getParameterTypes();
632                 String[] values = parameters.split(COMMA, -1);
633
634                 if (values.length == 1 && values[0].trim().equals(EMPTY)) {
635                     values = new String[0];
636                 }
637
638                 if (values.length != paramterTypes.length) {
639                     throw new MethodException(PARAMETER_COUNT_ERROR);
640                 }
641
642                 for (int i = 0; i < values.length; i++) {
643                     validAssigningPossibilities(paramterTypes[i], values[i].trim());
644                 }
645
646                 return;
647             }
648
649             throw new MethodException(METHOD_ERROR);
650         }
651
652         /**
653          * @return the global block.
654          */
655         public Block getGlobalBlock() {
656             Block block = this;
657
658             while (block.getFather() != null) {
659                 block = block.getFather();
660             }
661             return block;
662         }
663
664         /**
665          * @return the methods of the global block.
666          */
667         protected ArrayList<Method> getGlobalMethods() {
668             return getGlobalBlock().getMethods();
669         }
670     }
```

4 oop/ex7/blocks/BlockException.java

```java
package oop.ex7.blocks;

import oop.ex7.main.IllegalCodeException;

public class BlockException extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public BlockException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

5 oop/ex7/blocks/ConditionFactory.java

```java
package oop.ex7.blocks;

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

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;


/**
 * class ConditionFactory
 */
public class ConditionFactory {
    public static final String BLOCK_NOT_EXIST= "Block is not exist";
    /**
     * Creates a new condition block.
     * @param nameOfBlock the name of the block
     * @param content the content of the block
     * @param block the father block.
     * @return a new condition block.
     * @throws IllegalCodeException
     */
    public static Conditions createCondition
    (String nameOfBlock,ArrayList<String> content,Block block ) throws IllegalCodeException {

        Pattern paternIf = Pattern.compile(RegexBox.IF_REGEX);
        Matcher matchIf = paternIf.matcher(nameOfBlock);
        Pattern paternWhile = Pattern.compile(RegexBox.WHILE_REGEX);

        Matcher matchWhile = paternWhile.matcher(nameOfBlock);

        if(matchIf.find()){
            return new If(nameOfBlock,block,content);
        }

        else if(matchWhile.find()){
            return new While(nameOfBlock,block,content);
        }
        else throw new BlockException(BLOCK_NOT_EXIST);

    }
}
```

6 oop/ex7/blocks/Conditions.java

```java
package oop.ex7.blocks;

import java.util.ArrayList;

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

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;

/**
 * class conditions
 */
public class Conditions extends InnerBlock {

    protected String condition;

    /**
     * constructor
     * @param nameOfBlock the name of the block.
     * @param father the block father
     * @param content the block content
     * @throws IllegalCodeException
     */
    public Conditions(String nameOfBlock, Block father, ArrayList<String> content) throws IllegalCodeException {
        super(nameOfBlock, father, content);
    }

    /**
     *
     * checks whether the boolean condition is valid and throws an exception if not.
     */
    public void isValidCondition() throws IllegalCodeException {

        Pattern paternCondition = Pattern.compile(RegexBox.CONDITION_REGEX);
        Matcher matchCondition = paternCondition.matcher(this.condition);
        if(matchCondition.matches()){

            validAssigningPossibilities(BOOL, this.condition);
        }
    }

    /**
     * @return the boolean condition.
     */
    public String getCondition() {
        return this.condition;
    }
}
```

7 oop/ex7/blocks/GlobalBlock.java

```java
package oop.ex7.blocks;

import java.util.ArrayList;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.main.Tools;

/**
 * class global block.
 */
public class GlobalBlock extends Block {

    boolean methodsOnly;

    /**
     * constructor
     * @param content the global block content.
     */
    public GlobalBlock(ArrayList<String> content) {
        super(EMPTY, null, content);
    }

    /**
     * return true iff a given line is legal and false otherwise.
     */
    protected boolean checkLineAllowed(String line) throws IllegalCodeException {
        switch (line) {
            case Tools.COMMENT:
            case Tools.EMPTY_LINE:
            case Tools.METHOD_REGEX:
                return true;

            case Tools.DECLARATION_ON_VAR:
            case Tools.DECLARATION_AND_ASSIGNINNG_IN_VAR:
                return !methodsOnly;

            default:
                throw new NoSuchLine(NO_SUCH_LINE);
        }
    }


    /**
     * sends method to parse.
     * @param methodsOnly a boolean var that says if a given line is a method or not.
     * @throws IllegalCodeException
     */
    public void parse(boolean methodsOnly) throws IllegalCodeException {
        this.methodsOnly = methodsOnly;
        parseMethod("", null, content, methodsOnly);
    }
}
```

8 oop/ex7/blocks/If.java

```java
package oop.ex7.blocks;
import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;


/**
 * class If.
 */
public class If extends Conditions{
    public static final String INVALID_BLOCK="Invalid if block" ;
    /**
     * constructor.
     * @param nameOfBlock the name of the block
     * @param containerBlock the father block
     * @param content the content of the block.
     * @throws IllegalCodeException
     */
    public If
    (String nameOfBlock, Block containerBlock, ArrayList<String> content)
            throws IllegalCodeException {

        super(nameOfBlock, containerBlock, content);

        Pattern paternIf = Pattern.compile(RegexBox.IF_REGEX);
        Matcher matchIf = paternIf.matcher(nameOfBlock);

        if (!matchIf.matches()) {
            throw new IfBlockException(INVALID_BLOCK);
        }

        this.condition = matchIf.group(1);
        super.isValidCondition();

    }
}
```

9 oop/ex7/blocks/IfBlockException.java

```java
package oop.ex7.blocks;

import oop.ex7.main.IllegalCodeException;

public class IfBlockException extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public IfBlockException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

10 oop/ex7/blocks/InnerBlock.java

```java
package oop.ex7.blocks;

import java.util.ArrayList;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.main.Tools;

/**
 *
 * class InnerBlock.
 * Represents any block that is not the global block : method block or condition(if or while)
 * block.
 */
public class InnerBlock extends Block {

    /**
     * constructor
     * @param nameOfBlock the name of th block.
     * @param father the father block.
     * @param content the content of the block.
     */
    public InnerBlock(String nameOfBlock, Block father,
            ArrayList<String> content) {
        super(nameOfBlock, father, content);
    }

    /**
     * return true iff the given line is a legal line and false otherwise.
     */
    protected boolean checkLineAllowed(String line) throws IllegalCodeException {
        switch (line) {
        case Tools.COMMENT:
        case Tools.EMPTY_LINE:
        case Tools.DECLARATION_ON_VAR:
        case Tools.ASSIGNING_IN_VAR:
        case Tools.DECLARATION_AND_ASSIGNINNG_IN_VAR:
        case Tools.IF_OR_WHILE:
        case Tools.RETURN_LINE:
        case Tools.METHOD_CALL:
            return true;

        default:
            throw new NoSuchLine(NO_SUCH_LINE);
        }
    }
}
```

11 oop/ex7/blocks/InvalidIfBlock.java

```java
package oop.ex7.blocks;

import oop.ex7.main.IllegalCodeException;

public class InvalidIfBlock extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public InvalidIfBlock(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

12 oop/ex7/blocks/Method.java

```java
package oop.ex7.blocks;

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

import oop.ex7.main.IllegalCodeException;
import oop.ex7.main.Variable;
import oop.ex7.regex.RegexBox;

/**
 *
 * class Method.
 * represents an method block.
 */
public class Method extends InnerBlock {

    public static final String COMMA = ",";
    private String returnedValueType;
    private String methodName;
    private String[] parameters;
    private String[] parameterTypes;
    public static final String METHOD_NAME= "Block is not exist";
    public static final String RETURN_VALUE_ERROR= "Wrong returned value";
    public static final String DUPLICATE_PARAM= "Block is not exist";
    public static final String WRONG_PARAM= "Block is not exist";
    public static final String WRONG_DEC= "Wrong method block declaration";
    /**
     * constructor
     * @param nameOfMethod the signature of the method.
     * @param father the father of the method
     * @param content the content of the method (all the lines in that block).
     * @throws IllegalCodeException
     */
    public Method(String nameOfMethod, Block father, ArrayList<String> content) throws IllegalCodeException {

        super(nameOfMethod, father, content);
        Pattern paternMethod = Pattern.compile(RegexBox.METHOD_REGEX);
        Matcher matchMethod = paternMethod.matcher(nameOfMethod);


        if(matchMethod.matches()){
            this.methodName = matchMethod.group(6);
            this.returnedValueType = Variable.normalizeReturnType(matchMethod.group(1));
        }
        else
            throw new MethodException(WRONG_DEC);

        String paramsRepresntaion=matchMethod.group(7);
        this.parameters=paramsRepresntaion.split(COMMA,-1);
        isMethodParametersValid();
    }

    /**
      * checks if the returned value type of the method is valid.
     * @throws WrongReturnedValueTypeException
     */
    public void isReturnedValueTypeValid() throws IllegalCodeException {
        Pattern paternMethod = Pattern.compile(RegexBox.RETURN_TYPE_OF_METHOD);
```

```
60          Matcher matchMethod = paternMethod.matcher(this.returnedValueType);
61          if(!matchMethod.find()){
62              throw new MethodException(RETURN_VALUE_ERROR);
63          }
64      }
65
66      /**
67       * checks if the method name is valid.
68       * @throws IllegalCodeException
69       */
70      public void isNameMethodValid() throws IllegalCodeException {
71          Pattern paternMethod = Pattern.compile(RegexBox.METHOD_NAME);
72          Matcher matchMethod = paternMethod.matcher(this.methodName);
73
74          if(!matchMethod.matches()){
75              throw new MethodException(METHOD_NAME);
76          }
77      }
78
79
80      /**
81       * @return true iff the parameters who were sent are valid(right order).
82       * @throws IllegalCodeException
83       */
84      public void isMethodParametersValid() throws IllegalCodeException {
85
86          if (this.parameters.length == 1 && this.parameters[0].equals(EMPTY)) {
87              this.parameterTypes = new String[0];
88              return;
89          }
90
91          this.parameterTypes = new String[this.parameters.length];
92          ArrayList<String> parameterNames = new ArrayList<String>();
93
94          for(int i = 0; i < this.parameters.length; i++){
95              String parameter = this.parameters[i];
96              Pattern paternParameter = Pattern.compile(RegexBox.DECLARATION_ON_VAR);
97              Matcher matchParameter = paternParameter.matcher(parameter);
98
99              if(parameter.equals(EMPTY) ||
100                 !matchParameter.matches()) {
101                 throw new MethodException(WRONG_PARAM);
102             }
103
104             Variable var=new Variable(this, matchParameter.group(1), matchParameter.group(4), true);
105
106             if (parameterNames.contains(var.getName())) {
107                 throw new MethodException(DUPLICATE_PARAM);
108             }
109
110             parameterNames.add(var.getName());
111
112             this.parameterTypes[i] = Variable.normalizeType(matchParameter.group(2) + matchParameter.group(3));
113             this.localVaribles.add(var);
114         }
115     }
116
117
118     /**
119      * @return the returned value of the method.
120      */
121     public String getReturnedValue() {
122         return this.returnedValueType;
123     }
124
125     /**
126      * @return the name of the method.
127      */
```

```
128         public String getMethodName() {
129             return this.methodName;
130         }
131 
132         /**
133          * @return an array of the method parameters types
134          */
135         public String[] getParameterTypes() {
136             return this.parameterTypes;
137         }
138     }
```

13 oop/ex7/blocks/MethodException.java

```java
package oop.ex7.blocks;

import oop.ex7.main.IllegalCodeException;



public class MethodException extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public MethodException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

14 oop/ex7/blocks/NoSuchBlockException.java

```java
package oop.ex7.blocks;

public class NoSuchBlockException {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

15 oop/ex7/blocks/NoSuchLine.java

```java
package oop.ex7.blocks;

import oop.ex7.main.IllegalCodeException;

public class NoSuchLine extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public NoSuchLine(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

16 oop/ex7/blocks/While.java

```java
package oop.ex7.blocks;

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

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;

/**
 *
 * class While
 * represents a while block.
 */
public class While extends Conditions  {
    public static final String WHILE_BLOCK_ERROR= "Invalid while block";
    /**
     * constructor.
     * @param nameOfBlock the name of the block.
     * @param containerBlock the father of the block.
     * @param content the content of the block.
     * @throws IllegalCodeException
     */
    public While(String nameOfBlock, Block containerBlock,ArrayList<String> content) throws IllegalCodeException {

        super(nameOfBlock, containerBlock, content);

        Pattern paternWhile = Pattern.compile(RegexBox.WHILE_REGEX);
        Matcher matchWhile = paternWhile.matcher(nameOfBlock);

        if (!matchWhile.matches()) {
            throw new WhileException(WHILE_BLOCK_ERROR);
        }

        this.condition = matchWhile.group(1);
        super.isValidCondition();
    }
}
```

17 oop/ex7/blocks/WhileException.java

```java
package oop.ex7.blocks;

import oop.ex7.main.IllegalCodeException;

public class WhileException extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public WhileException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

18 oop/ex7/blocks/WrongNumberOfException.java

```java
package oop.ex7.blocks;

import oop.ex7.main.IllegalCodeException;

public class WrongNumberOfException  extends IllegalCodeException{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public WrongNumberOfException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

19 oop/ex7/main/DuplicateDeclaration.java

```java
package oop.ex7.main;

public class DuplicateDeclaration  extends IllegalCodeException{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public DuplicateDeclaration(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

20 oop/ex7/main/IllegalCodeException.java

```java
package oop.ex7.main;

public class IllegalCodeException extends Exception {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    /**
     *
     * constructor
     */
    public IllegalCodeException (String errorMassage) {

        System.out.println(1);
        System.out.println(errorMassage);
    }
}
```

21 oop/ex7/main/InvalidDeclaration.java

```java
package oop.ex7.main;

public class InvalidDeclaration extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public InvalidDeclaration(String errorMassage) {
        super(errorMassage);

    }

}
```

22 oop/ex7/main/InvalidValue.java

```java
package oop.ex7.main;

public class InvalidValue extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public InvalidValue(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

23 oop/ex7/main/NotInitlizedVariableException.java

```java
package oop.ex7.main;

public class NotInitlizedVariableException
extends IllegalCodeException  {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public NotInitlizedVariableException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

24 oop/ex7/main/Sjavac.java

```java
package oop.ex7.main;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Scanner;

import oop.ex7.blocks.GlobalBlock;
import oop.ex7.blocks.Method;
import oop.ex7.blocks.WrongNumberOfException;

/**
 * class Sjavac
 */
public class Sjavac {
    public static final String SUCCESES = "0";
    public static final String FAIL = "1";
    public static final String ILLGAL_INPUT= "2";
    public static final String WRONG_NUMBER_OF_BRACKET=
            "Wrong number of brackets";
    /**
     * main program.
     * @param args the sjava file.
     */
    public static void main (String args[]) {
        String sjava = args[0];

        try {

            ArrayList<String> file = readFile(sjava);

            try {
                toParse(file);
                System.out.println(SUCCESES);

            }
            catch(IllegalCodeException ex) {
                System.out.println(FAIL);
            }

        }
        catch(IOException ex) {
            System.out.println(ILLGAL_INPUT);
        }
        catch (Exception ex) {
            ByteArrayOutputStream baos = new java.io.ByteArrayOutputStream();
            PrintStream ps = new PrintStream(baos);
            ex.printStackTrace(ps);
            String stackTrace = baos.toString();
            System.out.print(stackTrace);
            System.out.println(ex.getMessage());
        }
    }


    /**
```

```
60          * reads the sjava file.
61          * @param sjava the sjava file.
62          * @return an array list of the sjava lines.
63          * @throws IOException
64          */
65         public static ArrayList<String> readFile(String sjava) throws IOException{
66 
67             ArrayList<String> file=new ArrayList<>();
68             File sourceFile = new File(sjava);
69 
70             Scanner sc = new Scanner(new FileInputStream(sourceFile));
71 
72 
73             while(sc.hasNextLine()) {
74                 String s = sc.nextLine();
75 
76                 file.add(s);
77             }
78             sc.close();
79             return file;
80         }
81 
82         /**
83          *
84          * @param file the sjava file in array list of strings. (any cell is a line).
85          * @throws IllegalCodeException
86          */
87         public static void toParse(ArrayList<String> file) throws IllegalCodeException {
88 
89             if (Tools.checkBracketsValidity(file)) { //the number of brackets is valid
90 
91 
92                 for(int i = 0; i< file.size(); i++) {
93                     Tools.replaceAllSpaces(file.get(i),i,file);
94 
95                     Tools.checkAndReplacePoints(file.get(i),file,i);
96 
97                 }
98 
99                 GlobalBlock block = new GlobalBlock(file);
100 
101                 block.parse(true);
102                 block.parse(false);
103 
104                 ArrayList<Method> methods = block.getMethods();
105 
106                 for (Method method : methods) {
107                     method.parseMethod(method.getReturnedValue(), null, method.getContent());
108                 }
109             }
110             else {
111                 throw new WrongNumberOfException(WRONG_NUMBER_OF_BRACKET);
112             }
113         }
114     }
```

25 oop/ex7/main/Tools.java

```java
package oop.ex7.main;

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

import oop.ex7.blocks.WrongNumberOfException;
import oop.ex7.regex.RegexBox;


/**
 * class Tools
 * Contains a useful methods which are used in many classes in the program.
 */
public class Tools {
    public static final String    POINT = ";";
    public static final String    EMPTY = "";
    public static final String    CLOSE_BRACKET = "}";
    public static final String    OPEN_BRACKET = "{";
    public static final int NOT_FOUND = -1;
    public static final int VALID_BRACKETS = 0;
    public static final String COMMENT = "comment";
    public static final String EMPTY_LINE = "empty line";
    public static final String DECLARATION_ON_VAR = "declaration on var";
    public static final String ASSIGNING_IN_VAR = "assigning in var";
    public static final String DECLARATION_AND_ASSIGNINNG_IN_VAR =
            "assign and declaration on var";
    public static final String IF_OR_WHILE = "if or while line";
    public static final String RETURN_LINE = "return";
    public static final String METHOD_REGEX = "method";
    public static final String METHOD_CALL = "method call";
    public static final String SPACE = " ";
    public static final String MISSING_POINT= "Missing ';' expression";
    public static final String NO_LINE= "No such kind of line";
    public static final String MISSING_CLOSE_BRACKET="Missing closing bracket";

    /**
     * checks which kind of line the given line is and throws an excption if
     * the line is not a legal line.
     * @param line a certain line.
     * @return a string that represents which line is the given line is.
     * @throws NoSuchLineException
     */
    public static String whichKindOfLine(String line) throws IllegalCodeException {

        Pattern paternComment = Pattern.compile(RegexBox.COMMENT_LINE);
        Matcher matchComment = paternComment.matcher(line);

        if (matchComment.matches()) {
            return COMMENT;
        }

        Pattern paternEmpty = Pattern.compile(RegexBox.EMPTY_LINE);
        Matcher matchEmpty = paternEmpty.matcher(line);

        if (matchEmpty.matches()) {
            return EMPTY_LINE;
        }

```

```
60          Pattern paternDeclareOnVar = Pattern.compile(RegexBox.DECLARATION_ON_VAR);
61          Matcher matchDeclareOnVar = paternDeclareOnVar.matcher(line);
62
63          if (matchDeclareOnVar.matches()) {
64
65              return DECLARATION_ON_VAR;
66          }
67          Pattern paternDeclareAndAssignOnVar =
68                  Pattern.compile(RegexBox.DECLARATION_AND_ASSIGING_ON_VAR);
69          Matcher matchDeclareAndAssignOnVar = paternDeclareAndAssignOnVar.matcher(line);
70
71          if (matchDeclareAndAssignOnVar.matches()) {
72
73              return DECLARATION_AND_ASSIGNINNG_IN_VAR;
74          }
75          Pattern paternAssignOnVar = Pattern.compile(RegexBox.ASSIGNING_ON_VAR);
76          Matcher matchAssignOnVar = paternAssignOnVar.matcher(line);
77          if (matchAssignOnVar.matches()) {
78              return ASSIGNING_IN_VAR;
79          }
80
81          Pattern paternIf = Pattern.compile(RegexBox.IF_REGEX);
82
83          Matcher matchIf = paternIf.matcher(line);
84          Pattern paternWhile = Pattern.compile(RegexBox.WHILE_REGEX);
85          Matcher matchWhile = paternWhile.matcher(line);
86          if (matchIf.matches() || matchWhile.matches()) {
87              return IF_OR_WHILE;
88          }
89
90          Pattern paternReturn = Pattern.compile(RegexBox.RETURN_LINE);
91          Matcher matchReturn = paternReturn.matcher(line);
92          if(matchReturn.matches()) {
93              return RETURN_LINE;
94          }
95
96          Pattern paternMethod = Pattern.compile(RegexBox.METHOD_REGEX);
97          Matcher matchMethod = paternMethod.matcher(line);
98          if(matchMethod.matches()) {
99
100             return METHOD_REGEX;
101         }
102
103         Pattern patternMethodCall = Pattern.compile(RegexBox.METHOD_CALL);
104         Matcher matchMethodCall = patternMethodCall.matcher(line);
105         if (matchMethodCall.matches()) {
106             return METHOD_CALL;
107         }
108
109         Pattern paternCloseBracket = Pattern.compile(RegexBox.CLOSE_BRACKET);
110         Matcher matchCloseBracket = paternCloseBracket.matcher(line);
111         if(matchCloseBracket.matches()) {
112
113             return CLOSE_BRACKET;
114         }
115         else {
116             return NO_LINE;
117         }
118     }
119
120
121     /**
122      * @return true iff the number of brackets in the block content
123      * is valid and false otherwise;
124      */
125     public static  boolean checkBracketsValidity(ArrayList<String> content) {
126         int counterOfbrackets = 0;
127
```

```
128         for(int i = 0; i < content.size(); i ++) {
129
130             if (content.get(i).contains(OPEN_BRACKET) &&
131                     !content.get(i).contains(CLOSE_BRACKET)) { //not an array declaration
132                 counterOfbrackets ++;
133             }
134             else if (content.get(i).contains(CLOSE_BRACKET) &&
135                     !content.get(i).contains(OPEN_BRACKET)) { //not an array declaration
136                 counterOfbrackets--;
137             }
138         }
139
140         if(counterOfbrackets == VALID_BRACKETS) {
141             return true;
142         }
143         else return false;
144     }
145
146
147
148     /**
149      * finds the content (all the lines) of the new block.
150      * @param startLine the first line of the block.
151      * @param indexOfLine the index of the start line.
152      * @return an array list which contains the block content.
153      */
154     public static ArrayList<String> findNewContent(ArrayList<String> content, String startLine, int indexOfLine) throws Ille
155
156         indexOfLine++; //start of block content
157         ArrayList<String> newContent = new ArrayList<>();
158
159         int counterBrackts = 1;
160
161         while(counterBrackts != 0) {
162             if (indexOfLine >= content.size()) {
163                 throw new WrongNumberOfException(MISSING_CLOSE_BRACKET);
164             }
165
166             if(content.get(indexOfLine).contains(OPEN_BRACKET) &&
167                     !content.get(indexOfLine).contains(CLOSE_BRACKET)) {
168
169                 counterBrackts ++;
170             }
171
172             else if (content.get(indexOfLine).contains(CLOSE_BRACKET)
173                     && !content.get(indexOfLine).contains(OPEN_BRACKET)) {
174
175                 counterBrackts --;
176
177             }
178             newContent.add(content.get(indexOfLine));
179             indexOfLine ++;
180         }
181         ArrayList<String> newContent1=new ArrayList<>();
182         for(int i=0;i<newContent.size()-1;i++){
183             newContent1.add(newContent.get(i));
184         }
185         return newContent1;
186     }
187
188     /**
189      * replace all the spaces of a given line in only one space.(normalize the line).
190      * @param line a given line.
191      * @param index the index of the given line.
192      * @param file a given file.
193      */
194     public static void replaceAllSpaces(String line,int index, ArrayList<String> file) {
195
```

```
196             Pattern paternDeclareAndAssignOnVar = Pattern.compile(RegexBox.MORE_THAN_ONE_SPACE);
197             Matcher matchDeclareAndAssignOnVar = paternDeclareAndAssignOnVar.matcher(line);
198             if(matchDeclareAndAssignOnVar.find()){
199                 line=matchDeclareAndAssignOnVar.replaceAll(SPACE);
200                 file.set(index, line);
201             }
202         }
203 
204 
205         /**
206          * checks if the given line should contains the ";" literal in the end of
207          * it and if so checks if it really contains that literal (throws exception if not)
208          * and replace the ";" in empty string.
209          * @param line a given line.
210          * @param file a given file.
211          * @param indexOfLine the index of the given lie.
212          * @throws IllegalCodeException
213          */
214         public static void checkAndReplacePoints
215         (String line, ArrayList<String> file ,int indexOfLine) throws IllegalCodeException {
216             Pattern paternReturn = Pattern.compile(RegexBox.RETURN_LINE);
217             Matcher matchReturn = paternReturn.matcher(line);
218             Pattern paternDeclare = Pattern.compile(RegexBox.DECLARATION_ON_VAR);
219             Matcher matchDeclare = paternDeclare.matcher(line);
220 
221             Pattern paternAssign = Pattern.compile(RegexBox.ASSIGNING_ON_VAR);
222             Matcher matchAssign = paternAssign.matcher(line);
223 
224             Pattern paternDeclareAndAssign =
225                     Pattern.compile(RegexBox.DECLARATION_AND_ASSIGING_ON_VAR);
226             Matcher matchDeclareAndAssign  = paternDeclareAndAssign.matcher(line);
227 
228             Pattern patternCall = Pattern.compile(RegexBox.METHOD_CALL);
229             Matcher matchCall = patternCall.matcher(line);
230 
231             if(matchDeclare.matches() || matchAssign.matches() || matchDeclareAndAssign.matches() ||matchReturn.matches() ||
232                     matchCall.matches()) {
233 
234                 if(line.contains(POINT)) {
235                     line = line.replaceAll(POINT, EMPTY);
236                     file.set(indexOfLine, line);
237                 }
238                 else {
239                     throw new IllegalCodeException(MISSING_POINT);
240                 }
241             }
242         }
243 }
```

26 oop/ex7/main/Variable.java

```java
package oop.ex7.main;

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

import oop.ex7.blocks.Block;
import oop.ex7.regex.RegexBox;
import oop.ex7.types.TypeException;
import oop.ex7.types.TypeFactory;

/**
 * class Variable.
 * represents a variable.
 */
public class Variable {
    public static final String BRACKET = "[]";
    public static final String OPEN_BRA = "[";
    public static final String TYPE_ERR = "Invalid type";
    private String type;
    private String name;
    private boolean value;
    private boolean isPrimitive = true;
    private Block block;


    /**
     * constructor.
     * @param block the block of the variable.
     * @param type the type of the variable.
     * @param name the name of the variable.
     * @param value is the value was initiallized or not.
     * @throws IllegalCodeException
     */
    public Variable(Block block, String type, String name, boolean value)
            throws  IllegalCodeException {
        this.block = block;
        this.value = value;
        this.type=type;
        this.name = name;

        fixArrayType();
    }

    /**
     * constructor.
     * @param block the block of the variable.
     * @param type the type of the variable.
     * @param name the name of the variable.
     * @throws IllegalCodeException
     */
    public Variable(Block block, String type,String name)
            throws IllegalCodeException {
        this(block, type, name, false);
    }

    /**
     * checks whether the the value and the type are vaild.
     * @param type a given type.
     * @param value a given value.
```

```
60          * @throws IllegalCodeException
61          */
62         public static void checkValidPrimitive(String type, String value)
63                 throws IllegalCodeException {
64             TypeFactory.createPrimitive(type, value);
65         }
66 
67         /**
68          * @return the variable block.
69          */
70         public Block getBlock() {
71             return block;
72         }
73 
74         /**
75          * sends the var type to normalization.
76          * @throws IllegalCodeException
77          */
78         private void fixArrayType() throws IllegalCodeException {
79             this.type = normalizeType(this.type);
80             this.isPrimitive = !this.type.contains(BRACKET);
81         }
82 
83         /**
84          *
85          * @param type var type.
86          * @return the normalized type according to regex valid_type.
87          * @throws IllegalCodeException
88          */
89         public static String normalizeType(String type) throws IllegalCodeException {
90             return normalizeType(type, RegexBox.VALID_TYPE);
91         }
92 
93         /**
94          *
95          * @param the var type.
96          * @return the normalized type according to regex return_type_of_method.
97          * @throws IllegalCodeException
98          */
99         public static String normalizeReturnType(String type) throws IllegalCodeException {
100            return normalizeType(type, RegexBox.RETURN_TYPE_OF_METHOD);
101        }
102 
103        /**
104         *
105         * @param type the var type.
106         * @param regex a given regex.
107         * @return the type + "[]" if the var is an array and the regular type if
108         * the var is not an array. throw exception if the type is invalid.
109         * @throws IllegalCodeException
110         */
111        private static String normalizeType(String type, String regex)
112                throws IllegalCodeException {
113            Pattern paternType = Pattern.compile(regex);
114            Matcher matchType = paternType.matcher(type);
115 
116            if(!matchType.matches()) {
117                throw new TypeException("Invalid type");
118            }
119 
120            type = matchType.group(1);
121 
122            if (type != null) {
123                if (matchType.group(2).contains(OPEN_BRA)) {
124                    type += BRACKET;
125                }
126            }
127            else {
```

```
128                 type = matchType.group(4);
129             }
130             return type;
131         }
132 
133         /**
134          * @return the var name.
135          */
136         public String getName() {
137             return this.name;
138         }
139 
140         /**
141          * @return the var type.
142          */
143         public String getType() {
144             return this.type;
145         }
146 
147         /**
148          * @return true if the value was initialized and false otherwise.
149          */
150         public boolean hasValue() {
151             return this.value;
152         }
153 
154         /**
155          * sets true in the value which means the value was initialized.
156          */
157         public void setValue() {
158             this.value = true;
159         }
160 
161         /**
162          * @return true if the variable is primitive and false otherwise( the var is an array).
163          */
164         public boolean getIsPrimitive(){
165             return isPrimitive;
166         }
167     }
```

27 oop/ex7/main/VariableNotExistException.java

```java
package oop.ex7.main;

public class VariableNotExistException extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public VariableNotExistException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

28 oop/ex7/regex/RegexBox.java

```java
package oop.ex7.regex;

/**
 * class RegexBox
 * All the regular expressions which were used in the program.
 */
public class RegexBox {

    public final static String VARIABLE_NAME = "([A-Za-z][_A-Za-z0-9]*|_[_A-Za-z0-9]+)";
    public final static String MINUS_VARIABLE_NAME ="(-)?\\s*" + VARIABLE_NAME;

    public final static String METHOD_NAME ="(([A-Za-z][A-Za-z0-9]*)\\s*\\((.*)\\))";

    public final static String METHOD_CALL ="\\s*" + METHOD_NAME + "\\s*;?\\s*";
    public final static String MINUS_METHOD_CALL ="\\s*(-)?" + METHOD_CALL;


    public final static String VALID_TYPE_WITH_SPACE = "\\s*(int|double|String|boolean|char)(\\s*\\[\\s*\\]\\s*|\\s+)";

    public final static String VALID_TYPE = "\\s*(int|double|String|boolean|char)((\\s*\\[\\s*\\]\\s*)|\\s*)";

    public final static String INTEGER_KIND_OF_TYPE = "-?\\s*"+VARIABLE_NAME+"\\s*|\\s*-?\\s*"+METHOD_NAME+"\\s*|\\s*-?\\s*[

    public final static String INTEGER_VALUE = INTEGER_KIND_OF_TYPE+"\\s*|"+INTEGER_KIND_OF_TYPE+"[+\\-*/]"+INTEGER_KIND_OF_

    public final static String DOUBLE_KIND_OF_TYPE = "\\s*-?\\s*(\\d+[.]?\\d*|[.]\\d+)\\s*";

    public final static String DOUBLE_VALUE = DOUBLE_KIND_OF_TYPE+"\\s*|"+DOUBLE_KIND_OF_TYPE+"[+\\-*/]"+DOUBLE_KIND_OF_TYPE

    public final static String CHAR_VALUE = "\\s*'.'\\s*";

    public final static String STRING_VALUE = "\\s*\".*\"\\s*";

    public final static String BOOLEAN_VALUE="\\s*(true|false)\\s*";

    public final static String METHOD_REGEX = "\\s*((" + VALID_TYPE_WITH_SPACE + ")|void\\s+)(([A-Za-z][_A-Za-z0-9]*)\\s*\\(

    public final static String DECLARATION_ON_VAR = "\\s*(" + VALID_TYPE_WITH_SPACE + ")" + VARIABLE_NAME + "\\s*(\\;)?";//t

    public final static String DECLARATION_AND_ASSIGING_ON_VAR="\\s*(" + VALID_TYPE_WITH_SPACE + ")" + VARIABLE_NAME + "\\s*

    public final static String COMMENT_LINE = "\\s*//(.*)";

    public final static String ALL="(.*)";

    public final static String EMPTY_LINE = "^\\s*$";

    public final static String RETURN_LINE ="\\s*(return)(\\s+(.*))?\\s*(\\;)?";

    public static final String ARRAY_VALUE_ELEMENT = DOUBLE_VALUE + "|" + VARIABLE_NAME;
    public static final String ARRAY_VALUE = "\\s*\\{\\s*(" + ARRAY_VALUE_ELEMENT + "(," + ARRAY_VALUE_ELEMENT + ")*)?\\s*\\

    public static final String ARRAY_ELEMENT = "\\s*" + VARIABLE_NAME + "\\s*\\[\\s*(" + "\\d+|-?\\d+\\s*[+\\-/\\*]\\s*-?\\d

    public static final String MINUS_ARRAY_ELEMENT = "\\s*(-)?" + ARRAY_ELEMENT;

    public final static String CONDITION_REGEX="\\s*(" + BOOLEAN_VALUE + "|"+METHOD_NAME+"|"+VARIABLE_NAME + "|" + ARRAY_ELE

    public final static String IF_REGEX = "\\s*if\\s*\\(" + CONDITION_REGEX + "\\)\\s*\\{\\s*";
```

```
60
61      public final static String WHILE_REGEX = "\\s*while\\s*\\(" + CONDITION_REGEX + "\\)\\s*\\{\\s*";
62
63      public final static String ASSIGNING_ON_VAR = "\\s*(" + VARIABLE_NAME + "|" + ARRAY_ELEMENT + ")\\s*=\\s*(.*)\\s*;?"; //
64
65      public static final String VALID_INDEX = "\\s*\\d+\\s*([+\\-/\\*]\\s*-?\\d+)?\\s*";
66
67      public static final String INTEGER_REGEX="\\s*-?\\s*\\d+((\\+|-|/|\\*)-?\\d+)?\\s*";
68
69      public static final String MORE_THAN_ONE_SPACE = "\\s+";
70
71      public static final String RETURN_TYPE_OF_METHOD= VALID_TYPE + "|\\s*(void)\\s*";
72
73      public static final String CLOSE_BRACKET = "\\}";
74
75      public static final String VAR_INSIDE_BRAKET_ARRAY="\\s*\\{(.*)\\}\\s*";
76
77      public final static String VALID_ARRAY_TYPE = "(int|double|String|boolean|char)\\s*\\[\\s*\\]";
78
79      public final static String DOUBLE_EXPRESSION = DOUBLE_KIND_OF_TYPE + "|" + MINUS_ARRAY_ELEMENT + "|\\s*-?\\s*" + METHOD_
80
81      public final static String DOUBLE_OPERATOR_EXPRESSION = "(" + DOUBLE_EXPRESSION + ")(\\s*[+\\-*/](" + DOUBLE_EXPRESSION
82
83      public final static String EXPRESSION_REGEX = "((" + DOUBLE_EXPRESSION + ")(\\s*[+\\-*/](" + DOUBLE_EXPRESSION + "))?)|(
84
85  }
```

29 oop/ex7/types/ArrayNotExistException.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;

public class ArrayNotExistException extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public ArrayNotExistException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

30 oop/ex7/types/BooleanVariable.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;

/**
 * class Boolean variable.
 * represents a variable of the type Boolean.
 */
public class BooleanVariable extends Type {

    /**
     * constructor.
     * @param value a boolean value.
     * @throws IllegalCodeException
     */
    public BooleanVariable(String value) throws IllegalCodeException {
        super(value);
    }

    /**
     * constructor.
     * @param value an array of a boolean values.
     * @throws IllegalCodeException
     */
    public BooleanVariable(String[] value) throws IllegalCodeException {
        super(value);
    }

    /**
     * checks whether the value is valid.
     */
    public void isValidValue(String value) throws IllegalCodeException {
        regex = RegexBox.CONDITION_REGEX;
         super.isValidValue(value);

    }
}
```

31 oop/ex7/types/CharacterVariable.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;

/**
 * class Character variable.
 * represents a variable of the type Character.
 */
public class CharacterVariable extends Type {

    /**
     * constructor.
     * @param value a char value.
     * @throws IllegalCodeException
     */
    public CharacterVariable(String value) throws IllegalCodeException {
        super(value);
    }

    /**
     * constructor.
     * @param values an array of char values.
     * @throws IllegalCodeException
     */
    public CharacterVariable(String[] values) throws IllegalCodeException {
        super(values);
    }

    /**
     * checks whether the value is valid.
     */
    public void isValidValue(String value) throws IllegalCodeException {
        regex = RegexBox.CHAR_VALUE;
         super.isValidValue(value);
    }
}
```

32 oop/ex7/types/DoubleVariable.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;

/**
 * class Double variable.
 * represents a variable of the type Double.
 */
public class DoubleVariable extends Type {

    /**
     * constructor.
     * @param value a double value.
     * @throws IllegalCodeException
     */
    public DoubleVariable(String value) throws IllegalCodeException {

        super(value);
        isValidValue(value);
    }

    /**
     * constructor.
     * @param values an array of Double values.
     * @throws IllegalCodeException
     */
    public DoubleVariable(String[] values) throws IllegalCodeException {
        super(values);

    }

    /**
     * checks whether the value is valid.
     */
    public void isValidValue(String value) throws IllegalCodeException {
        regex=RegexBox.DOUBLE_VALUE;

         super.isValidValue(value);
    }
}
```

33 oop/ex7/types/IntegerVariable.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;

/**
 * class Integer variable.
 * represents a variable of the type integer.
 */
public class IntegerVariable extends Type {

    /**
     * constructor.
     * @param value a given integer value.
     * @throws IllegalCodeException
     */
    public IntegerVariable(String value) throws IllegalCodeException {

        super(value);
    }

    /**
     * constructor.
     * @param values an array of an integer values.
     * @throws IllegalCodeException
     */
    public IntegerVariable(String[] values) throws IllegalCodeException {

        super(values);
    }


    /**
     * checks whether the value is valid.
     */
    public void isValidValue(String value) throws IllegalCodeException {

        regex = RegexBox.INTEGER_REGEX;

         super.isValidValue(value);
    }
}


```

34 oop/ex7/types/NoSuchType.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;

public class NoSuchType extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public NoSuchType(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

35 oop/ex7/types/StringVariable.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;
import oop.ex7.regex.RegexBox;

/**
 * class String value.
 * represents a variable of the type string.
 */
public class StringVariable extends Type {

    /**
     * constructor.
     * @param value a certain string value.
     * @throws IllegalCodeException
     */
    public StringVariable(String value) throws IllegalCodeException {
        super(value);
    }

    /**
     * constructor
     * @param value an array of string values.
     * @throws IllegalCodeException
     */
    public StringVariable(String[] value) throws IllegalCodeException {
        super(value);

    }

    /**
     * checks whether the value is valid.
     */
    public void isValidValue(String value) throws IllegalCodeException {
        regex = RegexBox.STRING_VALUE;
        super.isValidValue(value);
    }

}
```

36 oop/ex7/types/Type.java

```java
package oop.ex7.types;

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

import oop.ex7.blocks.Block;
import oop.ex7.main.IllegalCodeException;

/**
 * class Type.
 * represents a Variable type.
 */
public class Type {
    public static final String EMPTY = "";
    private String value;
    private String[]  values;
    public String regex=null;
    protected Block block;

    /**
     * constructor.
     * @param value a given value.
     * @throws IllegalCodeException
     */
    public  Type (String value) throws IllegalCodeException {
        this.value = value;

        isValidValue(value);
    }

    /**
     * @return the value.
     */
    public String getValue(){
        return value;
    }

    /**
     * constructor.
     * @param values an array of values.
     * @throws IllegalCodeException
     */
    public Type(String[] values) throws IllegalCodeException {

        this.values = values;

        if(!this.values[0].equals(EMPTY)) {

            for(int i=0; i < this.values.length; i++){

                isValidValue(values[i]);
            }
        }
    }

    /**
     * checks whether a given value is valid and throws an exception if not.
     * @param value a given value.
     * @throws IllegalCodeException
```

```
60          */
61         public  void isValidValue(String value) throws IllegalCodeException {
62 
63             Pattern patern = Pattern.compile(regex);
64             Matcher match = patern.matcher(value);
65 
66             if(!match.matches()){
67 
68                 throw new IllegalCodeException("Wrong value");
69             }
70         }
71     }
```

37 oop/ex7/types/TypeException.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;

public class TypeException extends IllegalCodeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public TypeException(String errorMassage) {
        super(errorMassage);
        // TODO Auto-generated constructor stub
    }

}
```

38 oop/ex7/types/TypeFactory.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;

/**
 * class Type factory.
 */
public class TypeFactory {
    public static final String NO_TYPE="Invalid initialization value";
    public static final String INT = "int";
    public static final String STRING = "String";
    public static final String CHAR = "char";
    public static final String BOOL = "boolean";
    public static final String DOUBLE = "double";
    /**
     * Creates a new type variables.
     * @param type the type of the var.
     * @param value the value of the var.
     * @return a new Variable with the relevant type.
     * @throws IllegalCodeException
     */
    public static Type createPrimitive(String type, String value) throws IllegalCodeException {
        switch (type) {
        case INT:
            return new IntegerVariable(value);
        case  DOUBLE:
            return new DoubleVariable(value);
        case CHAR:
            return new CharacterVariable(value);
        case BOOL:
            return new BooleanVariable(value);
        case STRING:
            return new StringVariable(value);
        }

        throw new NoSuchType(NO_TYPE);
    }
}
```

39 oop/ex7/types/WrongValueException.java

```java
package oop.ex7.types;

import oop.ex7.main.IllegalCodeException;

public class WrongValueException extends IllegalCodeException {

    /**
     *
     * @param errorMassage
     */
    public WrongValueException(String errorMassage) {
        super(errorMassage);
    }

    /**
     *
     */
    private static final long serialVersionUID = 1L;
}
```