Contents

1 Basic Test Results

```
1   Logins: liat_gin
2
3
4
5   compiling with
6        javac -cp .:/cs/course/2013/oop/lib/junit4.jar  *.java oop/ex7/main/*.java
7
8
9   tests output :
10           Perfect!
```

2 README

```
 1  liat_gin
 2  oranpa
 3
 4  ===========
 5  EX7 README:
 6  ==========
 7
 8  ============================
 9  Description of the classes:
10  ============================
11
12
13
14  oop.ex7.blocks:
15  ===============
16
17  1) Block: an abstract class that represents any kind of block in the program.
18     Any block starts with "{" and ends with: "}" (not includes arrays).
19     Block can be: The whole program, method and if or while.
20     Contains the abstract method "checkLineAllowed" which decides whether
21     the given line in the block is legal line or not. Therefore any class
22     that extends from block has to implement
23     this method.
24
25
26
27  2) ConditionFactory: class that creates a new condition block :
28     If block or while block.
29
30  3) Conditions: the father class of If and While.Represents block of the
31     type condition. Extends from InnerBlock Class.
32
33  4) GlobalBlock: represents the global block which means the block of
34     the whole file (if we think about it as a class so globalBlock is the all class).
35     Extends from Block.
36
37  5) If: represents a block of If. Extends from Condition.
38
39  6) InnerBlock: represents any block that is not the global block which includes
40     methods and condition blocks (if and while). Extends from Block.
41
42  7) Method: represents a method block. Extends from InnerBlock.
43
44  8) While: represents a while block. Extends from Condition.
45      EXCEPTIONS:
46      BlockException.java- thrown when there is a problem in the block.
47      IfBlockException.java- thrown when there is a problem with the if block.
48      MethodException.java-thrown when there is a problem with the method.
49      NoSuchBlock-thrown if the method doesnt exist.
50      NoSuchLine-thrown when the line is not exist.
51      WrongNumberOfException-throw when there is problem with
52      the number of bracket.
53
54
55
56
57  oop.ex7.main:
58  =============
59
```

```
60  9) IllegalCodeException: exception that being called any time the code in
61     sjavac is not a legal code. prints an informative massage about the specific
62     error that occurred.
63
64  10) Sjavac: the "main" class that contains the main method. In that class the first
65      reading from the file is happening and this is the first pass on the file which
66      "cleans" the file from comments, ";" literals and checks brackets validity.
67      After the first read sends the file to pars process (which parse the methods
68      and the all other blocks in the file).
69
70  11) Tools: this class contains useful methods that being used several times
71      in the program.
72
73  12) Variable: represents a Variable in the program. It can be a primitive
74      variable or not primitive variable (array). Contains all the essential
75      data on the variable.
76      EXCEPTIONS:
77      DuplicateDeclaration.java-exception thrown when duplicate declaration
78         IllegalCodeException.java- thrown whenever there is a problem in the file
79         InvalidValue.java-thrown if we put illegal value.
80         NoInitilizedVariableException.java-thrown if the var is not Initilized.
81         VariableNotExist.java-thrown if the var is not exist.
82
83
84  oop.ex7.regex:
85  ==============
86
87  13) RegexBox: this class contains all the regular expressions that we used
88      in the program.
89
90
91  oop.ex7.types:
92  ==============
93  14) Type: represents the type of the variable in the program.
94
95  15) TypeFactory: creates a new Variable according to the given type.
96
97  16) IntegerVariable: represents an integer type of value. Extends from Type.
98
99  17) DoubleVariable: represents an double type of value. Extends from Type.
100
101 18) BooleanVariable: represents an boolean type of value. Extends from Type.
102
103 19) StringVariable: represents an string type of value. Extends from Type.
104
105 20) CharVariable: represents an char type of value. Extends from Type.
106
107 21) WrongValueException: exception that being called when the value is not a
108     legal value. Extends from IllegalCodeException.
109  EXCEPTIONS:
110  TypeException.java-thrown if there is a problem with the type.
111  NoSuchType-thrown if we put invalid type to the var.
112
113 =====================
114 General desciption:
115 =====================
116
117 1. The design of ex7 : our code divided to 4 modules.
118
119 The first module is the main.
120 It's the module that suppose to connect between the other modules. Its class
121 contains the sjavac which go over the file, first it read it to list of lines
122 and then parse the file by creating a global block which is the file
123 itself and divide the file into methods and global variables.
124 The other classes are class called Tools which contain some methods that been used
125 for the entire program.
126
127 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.
     */
```

```java
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
```

```java
        if (Tools.checkBracketsValidity(content)) { //the number of brackets is valid

            for(int i = 0; i < content.size(); i ++) {

                String line = content.get(i);

                line = Tools.whichKindOfLine(content.get(i));

                if (!checkLineAllowed(line)) {
                    // Skip this line
                    continue;
                }

                switch (line) {

                case Tools.COMMENT:
                case Tools.EMPTY_LINE:
                    break;

                case Tools.METHOD_REGEX : {

                    ArrayList<String> methodContent =
                            Tools.findNewContent(content, content.get(i), i);
                    Method method = new Method(content.get(i), this, methodContent);

                    if (addMethods) {
                        methods.add(method);
                    }

                    i = i + methodContent.size() + 1;

                    break;
                }

                case Tools.DECLARATION_ON_VAR : {

                    Pattern paternDeclareOnVar =
                            Pattern.compile(RegexBox.DECLARATION_ON_VAR);
                    Matcher matchDeclareOnVar =
                            paternDeclareOnVar.matcher(content.get(i));

                    if (!matchDeclareOnVar.matches()) {
                        throw new InvalidDeclaration(INVALID_DEC);
                    }

                    Variable newVar =
                            new Variable
                            (this, matchDeclareOnVar.group(1)
                                    ,matchDeclareOnVar.group(4));
                    if(isNonGlobalVarExist(newVar.getName()) != null) {
                        throw new DuplicateDeclaration(DUPLICATE_VAR);
                    }
                    this.localVaribles.add(newVar);
                    break;
                }

                case Tools.ASSIGNING_IN_VAR : {
                    Pattern paternAssignOnVar =
                            Pattern.compile(RegexBox.ASSIGNING_ON_VAR);
                    Matcher matchAssignOnVar =
                            paternAssignOnVar.matcher(content.get(i));
                    if(!matchAssignOnVar.matches()){
                        throw new InvalidValue(INVALID_ASSIGN);
                    }
                    String nameOfVar = matchAssignOnVar.group(2);//splitedLine[0];
                    String value = matchAssignOnVar.group(7);//splitedLine[1];
                    String type = null;
```

```java
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));
```

```java
264                        if(matchCall.matches()){
265                            checkMethodCall(null, matchCall.group(2), matchCall.group(3));
266                        }
267                    }
268                    case Tools.RETURN_LINE : {
269                        String returnedvar = null;
270                        Pattern paternReturn = Pattern.compile(RegexBox.RETURN_LINE);
271                        Matcher matchReturn = paternReturn.matcher(content.get(i));
272                        if(matchReturn.matches()){
273                            returnedvar = matchReturn.group(3);
274                        }
275
276                        if(returnedvar!=null) {
277                            validAssigningPossibilities(returnedValueType, returnedvar);
278                        }
279                        break;
280                    }
281
282                    default:
283                        throw new NoSuchLine(NO_SUCH_LINE);
284                    }
285                }
286            }
287            else {
288                throw new WrongNumberOfException(WRONG_NUMBER_OF_BRACKET);
289            }
290    }
291
292
293        /**
294         * checks whether a new assigning is possible according to the types of the new
295         * assisgment value and the variable to assign to.
296         * @param destination the type of the var to assign into him.
297         * @param source the type of the new variable to assign into destination variable.
298         * @return true iff the types of the destination and source are equals or if destination
299         * is double and source is int. return false otherwise.
300         */
301        private boolean canAssign(String destination, String source) {
302            destination = destination.trim();
303            source = source.trim();
304
305            if (destination.equals(source) ||
306                    (destination.equals(DOUBLE) && source.equals(INT))) {
307                return true;
308            }
309            return false;
310        }
311
312
313        /**
314         * @param nameOfVar variable to check.
315         * @return the variable if its not a global variable
316         * and null if its a global variable.
317         */
318        public  Variable isNonGlobalVarExist(String nameOfVar) {
319            Variable var = isVarExist(nameOfVar);
320
321            if (var != null && var.getBlock().getFather() == null) {
322                var = null;
323            }
324
325            return var;
326        }
327
328        /**
329         * search for a given name of a variable.
330         * @param nameOfVar the name of the variable to search for.
331         * @return the variable itself if it exist and null otherwise.
```

```java
        */
    public  Variable isVarExist(String nameOfVar) {

        Pattern paternArrayElement = Pattern.compile(RegexBox.ARRAY_ELEMENT);
        Matcher matchArrayElement = paternArrayElement.matcher(nameOfVar);

        if(matchArrayElement.find()){
            nameOfVar=matchArrayElement.group(1);
        }

        Block block = this;

        while (block != null) {
            ArrayList<Variable> localVariables = block.getLocalVariables();

            int indexOfVar = findIndexOf(nameOfVar, localVariables);

            if(indexOfVar != NOT_FOUND) {
                return localVariables.get(indexOfVar);
            }

            block = block.getFather();
        }
        return null;
    }

    /**
     *
     * @param values a string of an array parameters.
     * @return an array list of strings that contains the
     * parameters of the array if exist.
     */
    public String[] splitArrayValue(String values) {
        Pattern paternVarInsideBraket = Pattern.compile
            (RegexBox.VAR_INSIDE_BRAKET_ARRAY);
        Matcher matchVarInsideBraket = paternVarInsideBraket.matcher(values);
        if(matchVarInsideBraket.matches()){
            return matchVarInsideBraket.group(1).split(COMMA, -1);
        }
        return null;

    }
    /**
     * @param var a variable to search for.
     * @param localVariables a list of the local variables that we have.
     * @return the index of the given variable if it was found in the local
     * variables list and -1 otherwise.
     */
    public  int findIndexOf(String nameOfVar, ArrayList<Variable> localVariable) {

        for (int i = 0; i < localVariable.size(); i++ ) {

            if(localVariable.get(i).getName().equals( nameOfVar) ) {
                return i;
            }
        }
        return NOT_FOUND;
    }

    /**
     * checks whether a certain assigning is legal and throws exception if
     * its not  legal assignment.
     * @param type the value type
     * @param value the value
     * @throws IllegalCodeException
     */
    public  void validAssigningPossibilities(String type, String value)
            throws IllegalCodeException {
```

```java
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) {
```

```java
            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]);
                }
```

```java
            }
        }
    }


    /**
     * checks whether a call to method is legal by check if this
     * method is already exist and if it exists
     * checks its return type and its parameters.
     * @param type the return type of the method.
     * @param name the name of the method.
     * @param parameters the parameters  of the method
     * @throws IllegalCodeException
     */
    protected void checkMethodCall(String type, String name, String parameters)
            throws IllegalCodeException {

        for(Method method : getGlobalMethods()){
            if (!method.getMethodName().equals(name)) {
                continue;
            }

            if(type != null && !canAssign(type, method.getReturnedValue())) {
                throw new TypeException( ILLEGAL_METHOD_RETURN_TYPE);
            }

            String[] paramterTypes = method.getParameterTypes();
            String[] values = parameters.split(COMMA, -1);

            if (values.length == 1 && values[0].trim().equals(EMPTY)) {
                values = new String[0];
            }

            if (values.length != paramterTypes.length) {
                throw new MethodException(PARAMETER_COUNT_ERROR);
            }

            for (int i = 0; i < values.length; i++) {
                validAssigningPossibilities(paramterTypes[i], values[i].trim());
            }

            return;
        }

        throw new MethodException(METHOD_ERROR);
    }

    /**
     * @return the global block.
     */
    public Block getGlobalBlock() {
        Block block = this;

        while (block.getFather() != null) {
            block = block.getFather();
        }
        return block;
    }

    /**
     * @return the methods of the global block.
     */
    protected ArrayList<Method> getGlobalMethods() {
        return getGlobalBlock().getMethods();
    }
}
```

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);
```

```java
            Matcher matchMethod = paternMethod.matcher(this.returnedValueType);
            if(!matchMethod.find()){
                throw new MethodException(RETURN_VALUE_ERROR);
            }
    }

    /**
     * checks if the method name is valid.
     * @throws IllegalCodeException
     */
    public void isNameMethodValid() throws IllegalCodeException {
        Pattern paternMethod = Pattern.compile(RegexBox.METHOD_NAME);
        Matcher matchMethod = paternMethod.matcher(this.methodName);

        if(!matchMethod.matches()){
            throw new MethodException(METHOD_NAME);
        }
    }


    /**
     * @return true iff the parameters who were sent are valid(right order).
     * @throws IllegalCodeException
     */
    public void isMethodParametersValid() throws IllegalCodeException {

        if (this.parameters.length == 1 && this.parameters[0].equals(EMPTY)) {
            this.parameterTypes = new String[0];
            return;
        }

        this.parameterTypes = new String[this.parameters.length];
        ArrayList<String> parameterNames = new ArrayList<String>();

        for(int i = 0; i < this.parameters.length; i++){
            String parameter = this.parameters[i];
            Pattern paternParameter = Pattern.compile(RegexBox.DECLARATION_ON_VAR);
            Matcher matchParameter = paternParameter.matcher(parameter);

            if(parameter.equals(EMPTY) ||
                !matchParameter.matches()) {
                throw new MethodException(WRONG_PARAM);
            }

            Variable var=new Variable(this, matchParameter.group(1), matchParameter.group(4), true);

            if (parameterNames.contains(var.getName())) {
                throw new MethodException(DUPLICATE_PARAM);
            }

            parameterNames.add(var.getName());

            this.parameterTypes[i] = Variable.normalizeType(matchParameter.group(2) + matchParameter.group(3));
            this.localVaribles.add(var);
        }
    }


    /**
     * @return the returned value of the method.
     */
    public String getReturnedValue() {
        return this.returnedValueType;
    }

    /**
     * @return the name of the method.
     */
```

```java
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());
        }
    }


    /**
```

```java
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;
        }

```

```java
        Pattern paternDeclareOnVar = Pattern.compile(RegexBox.DECLARATION_ON_VAR);
        Matcher matchDeclareOnVar = paternDeclareOnVar.matcher(line);

        if (matchDeclareOnVar.matches()) {

            return DECLARATION_ON_VAR;
        }
        Pattern paternDeclareAndAssignOnVar =
                Pattern.compile(RegexBox.DECLARATION_AND_ASSIGING_ON_VAR);
        Matcher matchDeclareAndAssignOnVar = paternDeclareAndAssignOnVar.matcher(line);

        if (matchDeclareAndAssignOnVar.matches()) {

            return DECLARATION_AND_ASSIGNINNG_IN_VAR;
        }
        Pattern paternAssignOnVar = Pattern.compile(RegexBox.ASSIGNING_ON_VAR);
        Matcher matchAssignOnVar = paternAssignOnVar.matcher(line);
        if (matchAssignOnVar.matches()) {
            return ASSIGNING_IN_VAR;
        }

        Pattern paternIf = Pattern.compile(RegexBox.IF_REGEX);

        Matcher matchIf = paternIf.matcher(line);
        Pattern paternWhile = Pattern.compile(RegexBox.WHILE_REGEX);
        Matcher matchWhile = paternWhile.matcher(line);
        if (matchIf.matches() || matchWhile.matches()) {
            return IF_OR_WHILE;
        }

        Pattern paternReturn = Pattern.compile(RegexBox.RETURN_LINE);
        Matcher matchReturn = paternReturn.matcher(line);
        if(matchReturn.matches()) {
            return RETURN_LINE;
        }

        Pattern paternMethod = Pattern.compile(RegexBox.METHOD_REGEX);
        Matcher matchMethod = paternMethod.matcher(line);
        if(matchMethod.matches()) {

            return METHOD_REGEX;
        }

        Pattern patternMethodCall = Pattern.compile(RegexBox.METHOD_CALL);
        Matcher matchMethodCall = patternMethodCall.matcher(line);
        if (matchMethodCall.matches()) {
            return METHOD_CALL;
        }

        Pattern paternCloseBracket = Pattern.compile(RegexBox.CLOSE_BRACKET);
        Matcher matchCloseBracket = paternCloseBracket.matcher(line);
        if(matchCloseBracket.matches()) {

            return CLOSE_BRACKET;
        }
        else {
            return NO_LINE;
        }
    }


    /**
     * @return true iff the number of brackets in the block content
     * is valid and false otherwise;
     */
    public static  boolean checkBracketsValidity(ArrayList<String> content) {
        int counterOfbrackets = 0;
```

```java
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>
155      findNewContent(ArrayList<String> content, String startLine, int indexOfLine)
156              throws IllegalCodeException {
157
158          indexOfLine++; //start of block content
159          ArrayList<String> newContent = new ArrayList<>();
160
161          int counterBrackts = 1;
162
163          while(counterBrackts != 0) {
164              if (indexOfLine >= content.size()) {
165                  throw new WrongNumberOfException(MISSING_CLOSE_BRACKET);
166              }
167
168              if(content.get(indexOfLine).contains(OPEN_BRACKET) &&
169                      !content.get(indexOfLine).contains(CLOSE_BRACKET)) {
170
171                  counterBrackts ++;
172              }
173
174              else if (content.get(indexOfLine).contains(CLOSE_BRACKET)
175                      && !content.get(indexOfLine).contains(OPEN_BRACKET)) {
176
177                  counterBrackts --;
178
179              }
180              newContent.add(content.get(indexOfLine));
181              indexOfLine ++;
182          }
183          ArrayList<String> newContent1=new ArrayList<>();
184          for(int i=0;i<newContent.size()-1;i++){
185              newContent1.add(newContent.get(i));
186          }
187          return newContent1;
188      }
189
190      /**
191       * replace all the spaces of a given line in only one space.(normalize the line).
192       * @param line a given line.
193       * @param index the index of the given line.
194       * @param file a given file.
195       */
```

```java
196        public static void replaceAllSpaces(String line,int index, ArrayList<String> file) {
197
198            Pattern paternDeclareAndAssignOnVar = Pattern.compile(RegexBox.MORE_THAN_ONE_SPACE);
199            Matcher matchDeclareAndAssignOnVar = paternDeclareAndAssignOnVar.matcher(line);
200            if(matchDeclareAndAssignOnVar.find()){
201                line=matchDeclareAndAssignOnVar.replaceAll(SPACE);
202                file.set(index, line);
203            }
204        }
205
206
207        /**
208         * checks if the given line should contains the ";" literal in the end of
209         * it and if so checks if it really contains that literal (throws exception if not)
210         * and replace the ";" in empty string.
211         * @param line a given line.
212         * @param file a given file.
213         * @param indexOfLine the index of the given lie.
214         * @throws IllegalCodeException
215         */
216        public static void checkAndReplacePoints
217        (String line, ArrayList<String> file ,int indexOfLine) throws IllegalCodeException {
218            Pattern paternReturn = Pattern.compile(RegexBox.RETURN_LINE);
219            Matcher matchReturn = paternReturn.matcher(line);
220            Pattern paternDeclare = Pattern.compile(RegexBox.DECLARATION_ON_VAR);
221            Matcher matchDeclare = paternDeclare.matcher(line);
222
223            Pattern paternAssign = Pattern.compile(RegexBox.ASSIGNING_ON_VAR);
224            Matcher matchAssign = paternAssign.matcher(line);
225
226            Pattern paternDeclareAndAssign =
227                    Pattern.compile(RegexBox.DECLARATION_AND_ASSIGING_ON_VAR);
228            Matcher matchDeclareAndAssign  = paternDeclareAndAssign.matcher(line);
229
230            Pattern patternCall = Pattern.compile(RegexBox.METHOD_CALL);
231            Matcher matchCall = patternCall.matcher(line);
232
233            if(matchDeclare.matches() || matchAssign.matches() || matchDeclareAndAssign.matches() ||matchReturn.matches() ||
234                    matchCall.matches()) {
235
236                if(line.contains(POINT)) {
237                    line = line.replaceAll(POINT, EMPTY);
238                    file.set(indexOfLine, line);
239                }
240                else {
241                    throw new IllegalCodeException(MISSING_POINT);
242                }
243            }
244        }
245    }
```

45

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.
```

```java
         * @throws IllegalCodeException
         */
        public static void checkValidPrimitive(String type, String value)
                throws IllegalCodeException {
            TypeFactory.createPrimitive(type, value);
        }


        /**
         * @return the variable block.
         */
        public Block getBlock() {
            return block;
        }


        /**
         * sends the var type to normalization.
         * @throws IllegalCodeException
         */
        private void fixArrayType() throws IllegalCodeException {
            this.type = normalizeType(this.type);
            this.isPrimitive = !this.type.contains(BRACKET);
        }


        /**
         *
         * @param type var type.
         * @return the normalized type according to regex valid_type.
         * @throws IllegalCodeException
         */
        public static String normalizeType(String type) throws IllegalCodeException {
            return normalizeType(type, RegexBox.VALID_TYPE);
        }


        /**
         *
         * @param the var type.
         * @return the normalized type according to regex return_type_of_method.
         * @throws IllegalCodeException
         */
        public static String normalizeReturnType(String type) throws IllegalCodeException {
            return normalizeType(type, RegexBox.RETURN_TYPE_OF_METHOD);
        }


        /**
         *
         * @param type the var type.
         * @param regex a given regex.
         * @return the type + "[]" if the var is an array and the regular type if
         * the var is not an array. throw exception if the type is invalid.
         * @throws IllegalCodeException
         */
        private static String normalizeType(String type, String regex)
                throws IllegalCodeException {
            Pattern paternType = Pattern.compile(regex);
            Matcher matchType = paternType.matcher(type);

            if(!matchType.matches()) {
                throw new TypeException("Invalid type");
            }

            type = matchType.group(1);

            if (type != null) {
                if (matchType.group(2).contains(OPEN_BRA)) {
                    type += BRACKET;
                }
            }
            else {
```

```java
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
    }

}
```

```java
1   package oop.ex7.regex;
2
3   /**
4    * class RegexBox
5    * All the regular expressions which were used in the program.
6    */
7   public class RegexBox {
8
9       public final static String VARIABLE_NAME = "([A-Za-z][_A-Za-z0-9]*|_[_A-Za-z0-9]+)";
10      public final static String MINUS_VARIABLE_NAME ="(-)?\\s*" + VARIABLE_NAME;
11
12      public final static String METHOD_NAME ="(([A-Za-z][A-Za-z0-9]*)\\s*\\((.*)\\))";
13
14      public final static String METHOD_CALL ="\\s*" + METHOD_NAME + "\\s*;?\\s*";
15      public final static String MINUS_METHOD_CALL ="\\s*(-)?" + METHOD_CALL;
16
17
18      public final static String VALID_TYPE_WITH_SPACE = "\\s*(int|double|String|boolean|char)"
19              + "(\\s*\\[\\s*\\]\\s*|\\s+)";
20
21      public final static String VALID_TYPE = "\\s*(int|double|String|boolean|char)"
22              + "((\\s*\\[\\s*\\]\\s*)|\\s*)";
23
24      public final static String INTEGER_KIND_OF_TYPE =
25              "-?\\s*"+VARIABLE_NAME+"\\s*|\\s*-?\\s*"+METHOD_NAME+"\\s*|\\s*-?\\s*[0-9]+\\s*";
26
27      public final static String INTEGER_VALUE =
28              INTEGER_KIND_OF_TYPE+"\\s*|"+INTEGER_KIND_OF_TYPE+"[+\\-*/]"+
29                  INTEGER_KIND_OF_TYPE + "\\s*";
30
31      public final static String DOUBLE_KIND_OF_TYPE =
32              "\\s*-?\\s*(\\d+[.]?\\d*|[.]\\d+)\\s*";
33
34      public final static String DOUBLE_VALUE =
35              DOUBLE_KIND_OF_TYPE+"\\s*|"+DOUBLE_KIND_OF_TYPE+"[+\\-*/]"+DOUBLE_KIND_OF_TYPE
36              + "\\s*";;
37
38          public final static String CHAR_VALUE = "\\s*'.'\\s*";
39
40          public final static String STRING_VALUE = "\\s*\".*\"\\s*";
41
42          public final static String BOOLEAN_VALUE="\\s*(true|false)\\s*";
43
44          public final static String METHOD_REGEX =
45                  "\\s*((" + VALID_TYPE_WITH_SPACE + ")|void\\s+)(([A-Za-z][_A-Za-z0-9]*)"
46                      + "\\s*\\((.*)\\))\\s*\\{\\s*";
47
48          public final static String DECLARATION_ON_VAR =
49                  "\\s*(" + VALID_TYPE_WITH_SPACE + ")" + VARIABLE_NAME + "\\s*(\\;)?";
50
51          public final static String DECLARATION_AND_ASSIGING_ON_VAR=
52                  "\\s*(" + VALID_TYPE_WITH_SPACE + ")" + VARIABLE_NAME + "\\s*=\\s*(.*)(\\;)?";
53
54          public final static String COMMENT_LINE = "\\s*//(.*)";
55
56          public final static String ALL="(.*)";
57
58          public final static String EMPTY_LINE = "^\\s*$";
59
```

```java
60          public final static String RETURN_LINE ="\\s*(return)(\\s+(.*))?\\s*(\\;)?";
61
62          public static final String ARRAY_VALUE_ELEMENT = DOUBLE_VALUE + "|" + VARIABLE_NAME;
63          public static final String ARRAY_VALUE = "\\s*\\{\\s*(" + ARRAY_VALUE_ELEMENT +
64                  "(," + ARRAY_VALUE_ELEMENT + ")*)?\\s*\\}\\s*";///to do
65
66          public static final String ARRAY_ELEMENT = "\\s*" + VARIABLE_NAME +
67                  "\\s*\\[\\s*(" + "\\d+|-?\\d+\\s*[+\\-/\\*]\\s*-?\\d+|" + MINUS_VARIABLE_NAME + ")\\s*\\]\\s*" ;
68
69          public static final String MINUS_ARRAY_ELEMENT = "\\s*(-)?" + ARRAY_ELEMENT;
70
71          public final static String CONDITION_REGEX="\\s*("
72                  + BOOLEAN_VALUE + "|"+METHOD_NAME+"|"+VARIABLE_NAME +
73                  "|" + ARRAY_ELEMENT + ")\\s*";
74
75          public final static String IF_REGEX = "\\s*if\\s*\\("
76                  + CONDITION_REGEX + "\\)\\s*\\{\\s*";
77
78          public final static String WHILE_REGEX = "\\s*while\\s*\\("
79                  + CONDITION_REGEX + "\\)\\s*\\{\\s*";
80
81          public final static String ASSIGNING_ON_VAR = "\\s*("
82                  + VARIABLE_NAME + "|" + ARRAY_ELEMENT + ")\\s*=\\s*(.*)\\s*;?";
83
84          public static final String VALID_INDEX = "\\s*\\d+\\s*([+\\-/\\*]\\s*-?\\d+)?\\s*";
85
86          public static final String INTEGER_REGEX="\\s*-?\\s*\\d+((\\+|-|/|\\*)-?\\d+)?\\s*";
87
88          public static final String MORE_THAN_ONE_SPACE = "\\s+";
89
90          public static final String RETURN_TYPE_OF_METHOD= VALID_TYPE + "|\\s*(void)\\s*";
91
92          public static final String CLOSE_BRACKET = "\\}";
93
94          public static final String VAR_INSIDE_BRAKET_ARRAY="\\s*\\{(.*)\\}\\s*";
95
96          public final static String VALID_ARRAY_TYPE =
97                  "(int|double|String|boolean|char)\\s*\\[\\s*\\]";
98
99          public final static String DOUBLE_EXPRESSION =
100                 DOUBLE_KIND_OF_TYPE + "|" + MINUS_ARRAY_ELEMENT + "|\\s*-?\\s*" +
101                     METHOD_NAME + "\\s*|\\s*-?\\s*" + VARIABLE_NAME + "\\s*";
102
103         public final static String DOUBLE_OPERATOR_EXPRESSION =
104                 "(" + DOUBLE_EXPRESSION + ")(\\s*[+\\-*/](" + DOUBLE_EXPRESSION + "))?";
105
106         public final static String EXPRESSION_REGEX =
107                 "((" + DOUBLE_EXPRESSION + ")(\\s*[+\\-*/](" + DOUBLE_EXPRESSION + "))?)"
108                     + "|(" + STRING_VALUE + ")|(" + CHAR_VALUE + ")|(" + BOOLEAN_VALUE + ")"
109                     + "|(" + ARRAY_VALUE + ")";
110
111  }
```

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

```
1   package oop.ex7.types;
2
3   import oop.ex7.main.IllegalCodeException;
4   import oop.ex7.regex.RegexBox;
5
6   /**
7    * class Character variable.
8    * represents a variable of the type Character.
9    */
10  public class CharacterVariable extends Type {
11
12      /**
13       * constructor.
14       * @param value a char value.
15       * @throws IllegalCodeException
16       */
17      public CharacterVariable(String value) throws IllegalCodeException {
18          super(value);
19      }
20
21      /**
22       * constructor.
23       * @param values an array of char values.
24       * @throws IllegalCodeException
25       */
26      public CharacterVariable(String[] values) throws IllegalCodeException {
27          super(values);
28      }
29
30      /**
31       * checks whether the value is valid.
32       */
33      public void isValidValue(String value) throws IllegalCodeException {
34          regex = RegexBox.CHAR_VALUE;
35           super.isValidValue(value);
36      }
37  }
```

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
```

```java
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;
}
```