# Contents

# 1 Basic Test Results

```
1  ******************** OOP pre-submission script for ex5 ********************
2
3  Extracting jar file...
4
5  Searching for file:  filesprocessing/DirectoryProcessor.java
6  Found file!
7  Searching for file:  README
8  Found file!
9  Checking README...
10
11
12
13 Compiling...
14
15
16 Running tests...
17     ===Executing test 002===
18 ===Executing test 007===
19 ===Executing test 019===
20 ===Executing test 021===
21     ===Executing test 030===
22 ===Executing test 047===
23 Perfect!
24
25
26 Checking efficiency of sort algorithm...
27     Excellent! Your sort algorithm is efficient enough.
28
29
```

# 2 README

```
brahan


============================
=      File description    =
============================
The jar file contains 1 package and the README file you are reading.
The package filesprocessing contain 2 subpackages,4 classes and 6 exception classes.
The package contain a file processing program.
The program gets a directory and a command File paths and will print out all the file in the
directory that hold the commands in the command file, and the warning if needed.
filters package contain all the needed files to filter the files,including filter method and
filter factory to create each filter according to the user input.
orders package contain all the needed files to sort the files, and classes which override the
java compare method, a order Factory for the orders so we will be able to create orders
according to the user input.
There are 2 types of exceptions Type one which only prints Warning messages and
Type two which will make the program to exit.

============================
=          Design          =
============================
2.The project design is very similar to take 2 in the uploaded Ex5-Design Suggestions and Extra
info ppt file in the module.
To my understanding take 2 was the most easy to understand between the three.
It kept the modularity principle to the maximum by separating each class into a different classes
and even packages (order/filter).
The flexibility of the code is also kept, if someone want to add new order/filter type all he need
to do is to add a class into their package and add a call to the new class into the package factory.
meaning there is no need to change any other classes.
By dividing the project into different classes the understandability of the code has raised.
The hierarchy of the project is:
DirectoryProcessor -->Manager-->Parser-->Section-->and he connect between the two packges.
the parser responsible on parsing the command file into sections
the sections responsible on creating the order and filter, using their factory's.
I used in the factorys a Decorator to implement the Negate operations.
============================
=  Implementation details   =
============================
filter package:
we have master filter which is a interface filter class(same as the ppt in the moodle)
each filter override the method filter which return a boolean value according to the needed
condition.
Order package:
each class override the compare method.
3.I chose to implement QuickSort, at the begging i implemented BubbleSort because its implementation
was the easiest, but after the mail from the staff i changed it into randomized QuickSort
as we learned in DAST its one of the most efficient sorting methods.
I sort the files in an arrayList of files so i dont need to worry about duplicate items,
because arrayList is not a set. furthermore the arraylist DS is very easy to mange
because there is methods which java alerdy implemented and to my understanding we can assume that
java will implement those methods much better then me.
we have a method in the Sorter class which wraps the sorting algorithm so the next user will be
able to change the sorting algorithm easily.
1.the section class will handle all the Type one errors raised by the packages, will print an
warning message if needed.
the parser method will handle all BadsubSection exception and some IOExceptions.
the manager will handle the IOExceptions and all the TypeTwoException we might have "missed"
thru program.
```

```
60  the DirectoryProcessor is the main class will handle the InvalidUsageExceptions
61  ============================
62  =    Answers to questions   =
63  ============================
64   Happy Jerusalem Day! :)
```

# 3 filesprocessing/BadCommandFileException.java

```
1  package filesprocessing;
2
3  /**
4   * the class represent the BadCommandFileException. represent all the problem with the commandFile
5   */
6  public class BadCommandFileException extends TypeTwoExceptions {
7
8      private static final String msg = "The Command File have a bad format \n";
9
10     /**
11      * constructor for the class, hold the needed msg for the exception
12      */
13     public BadCommandFileException() {
14         super(msg);
15     }
16
17 }
```

# 4 filesprocessing/BadSubSectionException.java

```java
package filesprocessing;

/**
 * will catch A bad sub-section name (i.e., not FILTER/ORDER). Sub-section names are case-sensitive
 */
class BadSubSectionException extends TypeTwoExceptions {
    private static final String msg = "Problem with subsections in CommandFile\n";

    /**
     * constructor for the class, hold the needed msg for the exception
     */
    BadSubSectionException() {
        super(msg);
    }

}
```

# 5 filesprocessing/DirectoryProcessor.java

```java
package filesprocessing;


import java.io.File;

/**
 * The main class for the program
 */
public class DirectoryProcessor {

    private static final int DIRECTORY_PLACE = 0;
    private static final int COMMAND_FILE_PLACE = 1;
    private static final int NUM_ARGS = 2;

    /**
     * the main mehtod for the project will handle the basic input Exceptions and then will call
     * the manager
     *
     * @param args the user input
     */
    public static void main(String[] args) {
        try {
            if (args.length != NUM_ARGS) {
                throw new InvalidUsageException();
            }
            File dir = new File(args[DIRECTORY_PLACE]);
            File commandFile = new File(args[COMMAND_FILE_PLACE]);
            if (!dir.isDirectory() || !commandFile.isFile()) {
                throw new IOException();
            }
            Manager manger = new Manager(dir, commandFile);
            manger.manageIt();
        } catch (InvalidUsageException | IOException e) {
            System.err.println(e.getMessage());
        }

    }


}
```

# 6 filesprocessing/IOException.java

```java
package filesprocessing;

/**
 * the class extends the TypeTwo exception handle IO exceptions
 */
class IOException extends TypeTwoExceptions {

    private static final String msg = "Problem with program IO\n";

    /**
     * constructor for the class, hold the needed msg for the exception
     */
    IOException() {
        super(msg);
    }
}
```

# 7 filesprocessing/InvalidUsageException.java

```java
package filesprocessing;

/**
 * the class extends the TypeTwo exception handle invalid usage exceptions
 */
class InvalidUsageException extends TypeTwoExceptions {
    private static final String msg = "Invalid usage Exception\n";

    /**
     * constructor for the class, hold the needed msg for the exception
     */
    InvalidUsageException() {
        super(msg);
    }
}
```

# 8 filesprocessing/Manager.java

```java
package filesprocessing;

import java.io.*;
import java.util.ArrayList;


/**
 * The class will mange the whole project, the class will take care of Type 2 Errors
 */
class Manager {
    private File[] directory;
    private File commandFile;

    Manager(File directory, File commandFile) {
        this.directory = directory.listFiles();
        this.commandFile = commandFile;
    }

    /**
     * The method responsible on manging the program will call the needed method from the project
     *
     * @throws IOException throwed if there were a problem accessing the commandFile
     */
    void manageIt() throws IOException {
        try {
            validateDirectory(this.directory);
            ArrayList<String> commandFile = Parser.parseCommandFile(this.commandFile);
            ArrayList<File> validDir = validateDirectory(this.directory);
            ArrayList<Section> sections = Parser.createSections(commandFile);
            for (Section section : sections) {
                ArrayList<File> preOutput = Parser.createOutput(validDir, section);
                for (File file : preOutput) {
                    System.out.println(file.getName());
                }
            }
        } catch (java.io.IOException e) {
            throw new IOException();
        } catch (TypeTwoExceptions typeTwoExceptions) {
            System.err.println(typeTwoExceptions.getMessage());
        }

    }


    /**
     * check if the directory is valid and then parse it
     *
     * @param directory the given directory
     * @return ArrayList of valid files in the directory
     */
    public ArrayList<File> validateDirectory(File[] directory) {
        ArrayList<File> fileList = new ArrayList<File>();
        if (directory != null) {
            for (File file : directory) {
                if (file.isFile()) {
                    fileList.add(file);
                }
            }
        }
```

```
60            return fileList;
61        }
62
63    }
```

# 9 filesprocessing/Parser.java

```java
package filesprocessing;


import filesprocessing.orders.*;
import filesprocessing.filters.*;

import java.io.*;

import java.util.*;
import java.util.Collections;


/**
 * The class will mange the section module, will be the only one that
 * knows the logical orders of the commands file
 */
public class Parser {

    private static final int SPECIAL_SECTION_SIZE = 3;
    private static final int DEFAULT_SECTION_SIZE = 4;
    private static final String FILTER = "FILTER";
    private static final String ORDER = "ORDER";
    private static final String DEFAULT_ORDER = "abs";
    private static final int SUBSECTION = 2;
    private static final int FILTER_PLACE = 1;
    private static final int ORDER_PLACE = 4;

    /**
     * the method will parse the command file and c
     *
     * @param commandFile blah blha
     * @return an array list representing the commandFile
     */
    static ArrayList<String> parseCommandFile(File commandFile) throws java.io.IOException {
        BufferedReader reader = new BufferedReader(new FileReader(commandFile));
        Object[] temp = reader.lines().toArray();
        String[] strings = Arrays.stream(temp).toArray(String[]::new);
        ArrayList<String> newList = new ArrayList<String>();
        Collections.addAll(newList, strings);
        return newList;
    }

    /**
     * the method responsible on creating the sections
     *
     * @param commandFile an array list representing the commandFile
     * @return will return an arrayList of sections
     * @throws BadSubSectionException will be thrown if the sections are not in a good format
     */
    static ArrayList<Section> createSections(ArrayList<String> commandFile)
            throws BadSubSectionException {
        ArrayList<Section> sections = new ArrayList<Section>();
        int i = 0;
        while (i < commandFile.size()) {
            if (commandFile.get(i).equals(FILTER)) {
                if (i + SPECIAL_SECTION_SIZE > commandFile.size() ||
                        commandFile.get(i + FILTER_PLACE) == null) {
                    throw new BadSubSectionException();
                }
```

```
60                  //size three section
61                  if ((commandFile.get(i + SUBSECTION).equals(ORDER)) &&
62                          (i + SPECIAL_SECTION_SIZE >= commandFile.size() ||
63                                  (commandFile.get(i + SPECIAL_SECTION_SIZE).equals(FILTER)))) {
64                      sections.add(new Section(commandFile.get(i + FILTER_PLACE), i + SUBSECTION,
65                              DEFAULT_ORDER, i + ORDER_PLACE));
66                      i += SPECIAL_SECTION_SIZE;
67                  } else if ((commandFile.get(i + SUBSECTION).equals(ORDER)) &&
68                          (i + DEFAULT_SECTION_SIZE >= commandFile.size() ||
69                                  (commandFile.get(i + DEFAULT_SECTION_SIZE).equals(FILTER)))) {
70                      //regular size section
71                      sections.add(new Section(commandFile.get(i + FILTER_PLACE), i + SUBSECTION,
72                              commandFile.get(i + SPECIAL_SECTION_SIZE), i + ORDER_PLACE));
73                      i += DEFAULT_SECTION_SIZE;
74                  } else {
75                      throw new BadSubSectionException();
76                  }
77              } else {
78                  throw new BadSubSectionException();
79              }
80          }
81          return sections;
82      }
83
84      /**
85       * the method will create the final program output, filtered and sorted arrayList
86       *
87       * @param dir     the given directory
88       * @param section the section according to the commandFile
89       * @return filtered and sorted Arraylist
90       */
91      static ArrayList<File> createOutput(ArrayList<File> dir, Section section) {
92          ArrayList<File> outputFile = new ArrayList<File>();
93          Filter filter = section.getFilter();
94          for (File file : dir) {
95              if (filter.filter(file)) {
96                  outputFile.add(file);
97              }
98          }
99          Sorter sortIt = new Sorter(outputFile, section.getOrder());
100         sortIt.sortIt();
101         return outputFile;
102     }
103
104
105 }
```

# 10 filesprocessing/Section.java

```java
package filesprocessing;

import filesprocessing.filters.*;
import filesprocessing.orders.*;


/**
 * the class will be responsible connecting between Order and Filter objects
 */
public class Section {

    private String filter;
    private String order;
    private int filterLine;
    private int orderLine;

    /**
     * constructor fot the section class
     *
     * @param filter     the filters in the section
     * @param filterLine the filters line in the commandFile, needed for the warning
     * @param order      the orders in the section
     * @param orderLine  the orders line in the commandFile, needed for the warning
     */
    Section(String filter, int filterLine, String order, int orderLine) {
        this.filter = filter;
        this.order = order;
        this.filterLine = filterLine;
        this.orderLine = orderLine;
    }

    /**
     * a getter method for the filters
     *
     * @return the filters using the filters factory, if an exception was raised in the "below"
     * classes will return the DEFAULT filters
     */
    public Filter getFilter() {
        try {
            return FilterFactory.filterFactory(this.filter);
        } catch (TypeOneException e) {
            e.msg(this.filterLine);
            return new All();

        }

    }

    /**
     * a getter method for the orders
     *
     * @return the needed orders using the orderFactory, an exception was raised in the "below"
     * * classes will return the DEFAULT orders
     */
    public Order getOrder() {
        try {
            return OrderFactory.orderFactory(this.order);
        } catch (TypeOneException e) {
            e.msg(this.orderLine);
```

```
60            }
61            return new OrderByName();
62
63        }
64
65
66    }
```

# 11 filesprocessing/TypeOneException.java

```java
package filesprocessing;

/**
 * Type1 exception, will handle all the little problems in the program- those problems
 * wont result the program to exit
 */
public class TypeOneException extends Exception {

    private static final String WARNING_MSG = "Warning in line ";

    /**
     * the method responsible on printing the warning
     *
     * @param line the line which the program has raised a problem
     */
    void msg(int line) {
        System.err.println(WARNING_MSG + line);
    }
}
```

# 12 filesprocessing/TypeTwoExceptions.java

```java
package filesprocessing;

/**
 * an abstract class for all the type2 exceptions
 */
abstract class TypeTwoExceptions extends Exception {

    private static final String ERROR = "ERROR: ";

    /**
     * constructor for the class, get the needed msg so we will be able to print it
     *
     * @param msg the needed msg, according to the exception
     */
    TypeTwoExceptions(String msg) {
        super(ERROR + msg);
    }
}
```

# 13 filesprocessing/filters/All.java

```
1
2  package filesprocessing.filters;
3
4  import java.io.File;
5
6  /**
7   * the class define the all filters, meaning we wont filters any files - defined as the
8   * DEFAULT filters for the program
9   */
10 public class All implements Filter {
11     /**
12      * @param file the file we want to check if its hold the condition
13      * @return true if the file hold the condition false otherwise
14      */
15     @Override
16     public boolean filter(File file) {
17         return true;
18     }
19 }
```

# 14 filesprocessing/filters/BetweenFilter.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the class represent the between filters, will filters all teh file that their size is not in the
 * given range
 */
class BetweenFilter extends FilterBySize {
    private double upperThreshold;
    private double lowerThreshold;

    /**
     * constructor for the between filters, will filters all the file that not in the given range
     *
     * @param lowerThreshold the lower bound for the file size given by the user
     * @param upperThreshold the upper bound for the file size given by the user
     */
    BetweenFilter(double lowerThreshold, double upperThreshold) {
        this.upperThreshold = upperThreshold;
        this.lowerThreshold = lowerThreshold;
    }

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return getSize(file) >= this.lowerThreshold
                && getSize(file) <= this.upperThreshold;
    }
}
```

# 15 filesprocessing/filters/Contains.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the class will filters the files by the given condition if they contain a certain value
 */
class Contains extends FilterByValue {
    private String value;

    /**
     * constructor for the contains filters, will get a string value which the files will be filtered
     * out if they not contain it
     */
    Contains(String value) {
        this.value = value;
    }

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return getFileName(file).contains(value);
    }
}
```

# 16 filesprocessing/filters/Executable.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the method represent the filters "executable"
 */
class Executable implements Filter {
    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return file.canExecute();
    }
}
```

# 17 filesprocessing/filters/FileName.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the class represent the filters "file"
 */
class FileName extends FilterByValue {

    private String value;

    /**
     * a  constructor for the Filter by file class
     *
     * @param file the file which we filters accordingly
     */
    FileName(String file) {
        this.value = file;
    }

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return getFileName(file).equals(value);
    }
}
```

# 18 filesprocessing/filters/Filter.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * Interface master class for the filters system
 */
public interface Filter {
    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    boolean filter(File file);
}
```

# 19 filesprocessing/filters/FilterBySize.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * an abstract class that hold method and variables relevant to all the filters
 * that need the file size
 */
abstract class FilterBySize implements Filter {

    private static final double KILOBYTE_FACTOR = 1024;

    /**
     * helper method to all the children class,will return the file size by KB
     *
     * @param file the file we want to check its size by KB
     * @return the file size in KB
     */
    double getSize(File file) {
        return file.length() / KILOBYTE_FACTOR;
    }


}
```

# 20 filesprocessing/filters/FilterByValue.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * an abstract class that hold all the methods that relevant to the filters by the string Value
 */
abstract class FilterByValue implements Filter {
    /**
     * helper method to all children class, return the file name
     *
     * @param file the file we want to check its name
     * @return the file name
     */
    String getFileName(File file) {
        return file.getName();
    }
}
```

# 21 filesprocessing/filters/FilterFactory.java

```java
package filesprocessing.filters;

// same as take2 in the ex5 suggested design

import filesprocessing.TypeOneException;

/**
 * Factory for the filters, will create a filters according to the user input
 */
public class FilterFactory {

    private static final String NEGATE = "NOT", ALL = "all", HIDDEN = "hidden",
            EXECUTABLE = "executable", WRITABLE = "writable",
            SUFFIX = "suffix", PREFIX = "prefix", CONTAINS = "contains", FILE_FILTER = "file",
            SMALLER_THAN = "smaller_than", BETWEEN = "between", GREATER_THAN = "greater_than",
            YES = "YES", NO = "NO";
    private static final String SEPARATOR = "#";
    private static final int FILTER_VAL = 0;
    private static final int FIRST_ARG = 1;
    private static final int SECOND_ARG = 2;
    private static final int MAX_ARG = 4;
    private static final int NEG_DEFAULT_ARG_NUM = 3;
    private static final int MIN_ARG_VAL = 0;
    private static final int DEFAULT_ARG_NUM = 2;

    /**
     * the method will create filters according to the given arguments
     *
     * @param input a string representing the wanted filters combination
     * @return the wanted filters
     * @throws TypeOneException if there is a problem with the given string will throw
     *                          an typeOneException as defined in the pdf
     */
    public static Filter filterFactory(String input) throws TypeOneException {
        String[] args = validateArguments(input);
        boolean isNot = false;
        if (input.substring(input.lastIndexOf(SEPARATOR) + 1).equals(NEGATE)) {
            isNot = true;
        }
        if ((args[FILTER_VAL].equals(WRITABLE) || args[FILTER_VAL].equals(EXECUTABLE) ||
                args[FILTER_VAL].equals(HIDDEN)) && (args[FIRST_ARG].equals(NO))) {
            isNot = true;
        }
        Filter result;
        switch (args[FILTER_VAL]) {
            case ALL:
                result = new All();
                return notFilterHandler(isNot, result);
            case HIDDEN:
                result = new Hidden();
                return notFilterHandler(isNot, result);
            case EXECUTABLE:
                result = new Executable();
                return notFilterHandler(isNot, result);
            case WRITABLE:
                result = new Writable();
                return notFilterHandler(isNot, result);
            case SUFFIX:
                result = new Suffix(args[FIRST_ARG]);
```

```
60                  return notFilterHandler(isNot, result);
61              case PREFIX:
62                  result = new Prefix(args[FIRST_ARG]);
63                  return notFilterHandler(isNot, result);
64              case CONTAINS:
65                  result = new Contains(args[FIRST_ARG]);
66                  return notFilterHandler(isNot, result);
67              case FILE_FILTER:
68                  result = new FileName(args[FIRST_ARG]);
69                  return notFilterHandler(isNot, result);
70              case SMALLER_THAN:
71                  result = new SmallerThanFilter(getDouble(args[FIRST_ARG]));
72                  return notFilterHandler(isNot, result);
73              case BETWEEN:
74                  result = new BetweenFilter(getDouble(args[FIRST_ARG]), getDouble(args[SECOND_ARG]));
75                  return notFilterHandler(isNot, result);
76              case GREATER_THAN:
77                  result = new GreaterThanFilter(getDouble(args[FIRST_ARG]));
78                  return notFilterHandler(isNot, result);
79              default:
80                  throw new TypeOneException();
81          }
82      }
83
84      /**
85       * the method will modify the given string into a manageable array
86       *
87       * @param filterString a string representing the wanted filters
88       * @return string list that each place describe an argument for the filters
89       */
90      private static String[] getArguments(String filterString) {
91          return filterString.split(SEPARATOR);
92      }
93
94      private static double getDouble(String value) throws TypeOneException {
95          double parsedDouble = Double.parseDouble(value);
96          if (parsedDouble < MIN_ARG_VAL) {
97              throw new TypeOneException();
98          } else {
99              return Double.parseDouble(value);
100         }
101     }
102
103     /**
104      * the method will validate the given arguments as defined in the pdf
105      *
106      * @param input the given string from the user
107      * @return will return a validated string array that represents the wanted filters
108      * @throws TypeOneException if the string is not valid will throw a typeOneException
109      */
110     private static String[] validateArguments(String input) throws TypeOneException {
111         String[] args = getArguments(input);
112         if (args.length > MAX_ARG) {
113             throw new TypeOneException();
114         } else if (args[FILTER_VAL].equals(BETWEEN) && args.length > DEFAULT_ARG_NUM &&
115                 getDouble(args[FIRST_ARG]) > getDouble(args[SECOND_ARG])) {
116             throw new TypeOneException();
117         } else if (!args[FILTER_VAL].equals(BETWEEN) && args.length == NEG_DEFAULT_ARG_NUM &&
118                 !input.substring(input.lastIndexOf(SEPARATOR) + FIRST_ARG).equals(NEGATE)) {
119             throw new TypeOneException();
120         } else if (args[FILTER_VAL].equals(BETWEEN) && args.length == MAX_ARG &&
121                 !input.substring(input.lastIndexOf(SEPARATOR) + FIRST_ARG).equals(NEGATE)) {
122             throw new TypeOneException();
123         } else if ((args[FILTER_VAL].equals(SMALLER_THAN) || args[FILTER_VAL].equals(GREATER_THAN))
124                 && args.length > NEG_DEFAULT_ARG_NUM) {
125             throw new TypeOneException();
126         } else if ((args[FILTER_VAL].equals((WRITABLE)) || args[FILTER_VAL].equals((EXECUTABLE))
127                 || args[FILTER_VAL].equals((HIDDEN))) && (!args[FIRST_ARG].equals(YES) &&
```

```
128                 !args[FIRST_ARG].equals(NO))) {
129             throw new TypeOneException();
130         }
131         return args;
132     }
133
134     /**
135      * will mange the negate decorator
136      *
137      * @param isNot  a boolean value will be true if the user input include NOT or NO
138      * @param filter the wanted filters
139      * @return a negated filters if needed
140      */
141     private static Filter notFilterHandler(boolean isNot, Filter filter) {
142         if (isNot) {
143             return new NegFilter(filter);
144         } else {
145             return filter;
146         }
147     }
148 }
```

# 22 filesprocessing/filters/GreaterThanFilter.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the class represent the filters "greater_than"
 */
class GreaterThanFilter extends FilterBySize {

    private double lowerThreshold;

    /**
     * a constructor for the greater than filters
     *
     * @param lowerThreshold the lower bound that all the file need to uphold
     */
    GreaterThanFilter(double lowerThreshold) {
        this.lowerThreshold = lowerThreshold;
    }

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return getSize(file) > lowerThreshold;
    }
}
```

# 23 filesprocessing/filters/Hidden.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * The class represents the filters "hidden"
 */
class Hidden implements Filter {
    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return file.isHidden();
    }
}
```

# 24 filesprocessing/filters/NegFilter.java

```
1  package filesprocessing.filters;
2
3  import java.io.File;
4
5  /**
6   * a decorator class, will get a given condition and negate its output
7   */
8  class NegFilter implements Filter {
9
10     private Filter condition;
11
12     /**
13      * constructor for the NegFilter class, will negate the given condition
14      *
15      * @param condition the condition which we want to negate
16      */
17     NegFilter(Filter condition) {
18         this.condition = condition;
19     }
20
21     /**
22      * @param file the file we want to check if its hold the condition
23      * @return true if the file hold the condition false otherwise
24      */
25     @Override
26     public boolean filter(File file) {
27         return !(this.condition.filter(file));
28     }
29 }
```

# 25 filesprocessing/filters/Prefix.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the class represent the filters "prefix"
 */
class Prefix extends FilterByValue {

    private String value;

    Prefix(String value) {
        this.value = value;
    }

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return getFileName(file).startsWith(this.value);
    }
}
```

# 26 filesprocessing/filters/SmallerThanFilter.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the class represent the filters "smaller_than"
 */
class SmallerThanFilter extends FilterBySize {

    private double upperThreshold;

    /**
     * a constructor for the greater than filters
     *
     * @param upperThreshold the upper bound that all the files need to uphold
     */
    SmallerThanFilter(double upperThreshold) {
        this.upperThreshold = upperThreshold;
    }

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return getSize(file) < this.upperThreshold;
    }
}
```

# 27 filesprocessing/filters/Suffix.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * the class represent the filters "suffix"
 */
class Suffix extends FilterByValue {

    private String value;

    /**
     * a constructor for the suffix filters
     *
     * @param value the suffix that we want to filters upon
     */
    Suffix(String value) {
        this.value = value;
    }

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return getFileName(file).endsWith(this.value);
    }
}
```

# 28 filesprocessing/filters/Writable.java

```java
package filesprocessing.filters;

import java.io.File;

/**
 * The class represent the "writable" filters
 */
class Writable implements Filter {

    /**
     * @param file the file we want to check if its hold the condition
     * @return true if the file hold the condition false otherwise
     */
    @Override
    public boolean filter(File file) {
        return file.canWrite();
    }
}
```

# 29 filesprocessing/orders/Order.java

```java
package filesprocessing.orders;

import java.io.File;
import java.util.Comparator;

/**
 * an abstract class that implements the orders for the project
 */
public abstract class Order implements Comparator<File> {

    static final String FILE_NAME_SEPARATOR = ".";

    /**
     * the method override the compare method in Comparator, compare by pre defined condition
     *
     * @param lhs the first argument
     * @param rhs the second argument
     * @return 1 if the left arg is bigger, -1 if the right arg is bigger,0 if equal
     */
    public abstract int compare(File lhs, File rhs);


}
```

# 30 filesprocessing/orders/OrderByName.java

```java
package filesprocessing.orders;

import java.io.File;

/**
 * this class override the java compare method by using jave compareTo method, will compare
 * the by their name, defined as the DEFAULT orders
 */
public class OrderByName extends Order {
    /**
     * the method override the compare method in Comparator, compare by pre defined condition-NAME
     *
     * @param lhs the first argument
     * @param rhs the second argument
     * @return 1 if the left arg is bigger, -1 if the right arg is bigger,0 if equal
     */
    @Override
    public int compare(File lhs, File rhs) {
        return lhs.getAbsolutePath().compareTo(rhs.getAbsolutePath());
    }
}
```

# 31 filesprocessing/orders/OrderBySize.java

```java
package filesprocessing.orders;

import java.io.File;

/**
 * this class override the java compare method by using jave compareTo method, will compare
 * the by their size,
 */
class OrderBySize extends Order {

    private static final int FIRST_ARG = 1;
    private static final int SECOND_ARG = -1;

    /**
     * the method override the compare method in Comparator, compare by pre defined condition-SIZE
     *
     * @param lhs the first argument
     * @param rhs the second argument
     * @return 1 if the left arg is bigger, -1 if the right arg is bigger,0 if equal
     */
    @Override
    public int compare(File lhs, File rhs) {
        if (lhs.length() > rhs.length()) {
            return FIRST_ARG;
        } else if (lhs.length() < rhs.length())
            return SECOND_ARG;
        else {
            return (lhs.getAbsolutePath().compareTo(rhs.getAbsolutePath()));

        }
    }
}
```

# 32 filesprocessing/orders/OrderByType.java

```java
package filesprocessing.orders;

import java.io.File;

/**
 * this class override the java compare method will compare the by their type
 */
class OrderByType extends Order {

    private static final int EQUAL_VAL = 0;

    /**
     * the method override the compare method in Comparator, compare by pre defined condition-TYPE
     *
     * @param lhs the first argument
     * @param rhs the second argument
     * @return 1 if the left arg is bigger, -1 if the right arg is bigger,0 if equal
     */
    @Override
    public int compare(File lhs, File rhs) {

        int compareVal = getFileType(lhs).compareTo(getFileType(rhs));
        if (compareVal == EQUAL_VAL) {
            return lhs.getName().compareTo(rhs.getName());
        } else {
            return compareVal;
        }

    }

    /**
     * helper method for the comapre by type class, will get us the file type
     *
     * @param file the file we want to get its type
     * @return a string representing the file type
     */
    private String getFileType(File file) {
        return file.getName().substring(file.getName().lastIndexOf(FILE_NAME_SEPARATOR) + 1);
    }
}
```

# 33 filesprocessing/orders/OrderFactory.java

```java
package filesprocessing.orders;

import filesprocessing.TypeOneException;

/**
 * a factory class for the orders class
 */
public class OrderFactory {

    private static final String SEPARATOR = "#";
    private static final String SIZE = "size", TYPE = "type", ABSOLUTE = "abs", REVERSE = "REVERSE";
    private static final int ORDER_VAL = 0;

    /**
     * the method responsible on creating all the needed orders according to the user input
     *
     * @param input the user input
     * @return an orders according to the user input
     * @throws TypeOneException if there is any problem with the user we throw an exception
     */
    public static Order orderFactory(String input) throws TypeOneException {
        String[] orderArgs = parseArguments(input);
        boolean isReverse = false;
        if (input.substring(input.lastIndexOf(SEPARATOR) + 1).equals(REVERSE)) {
            isReverse = true;
        }
        Order result;
        switch (orderArgs[ORDER_VAL]) {
            case SIZE:
                result = new OrderBySize();
                return reverseHandler(isReverse, result);
            case TYPE:
                result = new OrderByType();
                return reverseHandler(isReverse, result);
            case ABSOLUTE:
                result = new OrderByName();
                return reverseHandler(isReverse, result);
            default:
                throw new TypeOneException();
        }
    }

    /**
     * parse the arguments from the given string into an array
     *
     * @param orderName the given string input
     * @return an string array which each cell represent argument value
     */
    private static String[] parseArguments(String orderName) {
        return orderName.split(SEPARATOR);
    }

    /**
     * handle the reverse decorator
     *
     * @param isReverse true if there were REVERSE in the right place in the user input,
     *                  otherwise false
     * @param order     the wanted orders
     * @return an orders according to if we need to reverse the file
```

```
60          */
61         private static Order reverseHandler(boolean isReverse, Order order) {
62             if (isReverse) {
63                 return new ReverseOrder(order);
64             } else {
65                 return order;
66             }
67         }
68     }
```

# 34 filesprocessing/orders/ReverseOrder.java

```java
package filesprocessing.orders;

import java.io.File;

/**
 * a decorator for the orders, will reverse the orders of the given orders
 */
class ReverseOrder extends Order {

    private static final int REVERSE_FACTOR = -1;
    private Order sequence;

    /**
     * constructor for the decorator class, will "negate" the given orders
     *
     * @param sequence the orders that the user want to reverse
     */
    ReverseOrder(Order sequence) {
        this.sequence = sequence;
    }

    /**
     * the method override the compare method in Comparator, compare by pre defined condition
     *
     * @param lhs the first argument
     * @param rhs the second argument
     * @return -1 if the left arg is bigger, 1 if the right arg is bigger,0 if equal
     */
    @Override
    public int compare(File lhs, File rhs) {
        return sequence.compare(lhs, rhs) * REVERSE_FACTOR;
    }
}
```

# 35 filesprocessing/orders/Sorter.java

```java
package filesprocessing.orders;

import java.io.File;
import java.util.ArrayList;
import java.util.Collections;

/**
 * The class will use the orders defined in this package and sort the file according to them
 */
public class Sorter {

    private static final int EQUAL_VAL = 0;
    private ArrayList<File> arrayList;
    private Order order;

    /**
     * constructor for the Sorter class, Wraps the sorting algorithm-in our case quickSort
     *
     * @param arrayList the array we want to sort
     * @param order     the orders we want our files will be sorted upon
     */
    public Sorter(ArrayList<File> arrayList, Order order) {
        this.arrayList = arrayList;
        this.order = order;
    }

    /**
     * the method sort the array using the sorting algorithm
     */
    public void sortIt() {
        quickSort(this.arrayList, 0, arrayList.size() - 1, this.order);
    }

    /**
     * helper method for the quickSort method, will divide the method with a random pivot point
     *
     * @param arrayList the arrayList we want to sort
     * @param start     the array first index
     * @param end       the array last index
     * @param order     the orders which the array will be sorted upon
     * @return an integer that represent the place that the other objects are sorted upon
     */
    private static int partition(ArrayList<File> arrayList, int start, int end, Order order) {
        int random = start + ((int) Math.random() * (arrayList.size())) / (end - start + 1);
        int last = end;
        Collections.swap(arrayList, random, end);
        end--;
        while (start <= end) {
            if (order.compare(arrayList.get(start), arrayList.get(last)) < EQUAL_VAL)
                start++;
            else {
                Collections.swap(arrayList, start, end);
                end--;
            }
        }
        Collections.swap(arrayList, start, last);
        return start;
    }

```

```
60      /**
61       * quickSort algorithm as we implemented as we learned in DAST
62       *
63       * @param arrayList the arrayList we want to sort
64       * @param start     the array first index
65       * @param end       the array last index
66       * @param order     the orders which the array will be sorted upon
67       */
68      public void quickSort(ArrayList<File> arrayList, int start, int end, Order order) {
69          if (start >= end) return;
70          if (start < 0) return;
71          if (end > arrayList.size() - 1) return;
72
73          int pivot = partition(arrayList, start, end, order);
74          quickSort(arrayList, start, pivot - 1, order);
75          quickSort(arrayList, pivot + 1, end, order);
76      }
77  }
78
79
80
81
```