Contents

1 Basic Test Results

```
1  ******************** OOP pre-submission script for ex5 ********************
2
3  Extracting jar file...
4
5  /tmp/bodek.HVPOqs/oops/ex5/bar246802/presubmission/testdir/17510
6  Searching for file:  filesprocessing/DirectoryProcessor.java
7  Found file!
8  Searching for file:  README
9  Found file!
10 Checking README...
11
12
13
14 Compiling...
15
16
17 Running tests...
18     ===Executing test 002===
19 ===Executing test 007===
20 ===Executing test 019===
21 ===Executing test 021===
22     ===Executing test 030===
23 ===Executing test 047===
24 Perfect!
25
26
27 Checking efficiency of sort algorithm...
28     Excellent! Your sort algorithm is efficient enough.
29
30
```

2 README

```
bar246802



============================
=      File description    =
============================
DirectoryProcessingExceptions.java - Class for exceptions related to the processing.
DirectoryProcessor.java - The main class running the files processing according to user's input.
FileInfo.java - class to store each file needed info for future usage.
FileInfoFactory.java - Class for retrieving the files' info from the source directory.
FileUtils.java - A utilities class for file manipulation.
ProcessOperation.java - Class to store processing actions from the commands file.
ProcessOperationCore.java - Class to store processing actions from the commands file.
ProcessOperationFactory.java - Class for retrieving the desired processing tasks from the commands file.
RegexUtils.java - Utils class for regex related methods.
FilesFilter.java - interface for Class for filtering the files array by some properties.
FilesFilterFactory.java - Class for creating a new filter object for the file processing operation.
FilterAll.java - Get all the files, no filter.
FilterByContains.java - Filter files by containing a value their name.
FilterByExecutable.java - Filter files by executable property.
FilterByHidden.java - Filter files by visibility.
FilterByName.java - Filter the files by their name.
FilterByPrefix.java - Filter the files by the prefix of their name.
FilterBySizeBTW.java - Filter files by who's size smaller than given value.
FilterBySizeGT.java - Filter the files who's size greater than given value.
FilterBySizeLT.java - Filter files by who's size smaller than given value.
FilterBySuffix.java - Filter files by the suffix of their name.
FilterByWriteable.java - Filter files by writable property.
FilterCommand.java - enum class for filters conts types
FilesOrder.java - abstract Class for sorting the files array by some properties.
FilesOrderFactory.java - Class for creating a new order object for the file processing operation.
OrderByName.java - Class for sorting the files array by name.
OrderBySize.java - Class for sorting the files array by size, if size eq than by name.
OrderByType.java - Class for sorting the files array by type, if type eq than by name.
OrderCommand.java - enum class for orders conts types.



============================
=          Design          =
============================
I designed this ex similar to ex2, with the spaceship games.
I created a main class - DirectoryProcessor.java - to receive the user's args
and then created another class to process the given info in the files & folder- ProcessOperationFactory.java.
Each filter type implements the FilesFilte interface - meaning each filter class
implements the filter method as the.
Similarly, for the order types I used an abstract class FilesOrder which every order-type extends
and implement the compare method.
For the sorting itself I used a quicksort algorithm.



============================
=  Implementation details   =
============================
I implemented this ex by first processing the folder's files and getting all the needed info about them.
```

Then I processed the Commands file and saved each section as a processing
operation object - ProcessOperation.java.
In the preparation process I used a few Factory classes both for the Filter and Order
types of the operation and for the file's info inside the source directory.
By creation classes for each type of filter/order we are now able to call the filter and sort methods just
like in the Animal Class in the lessens when we were able to call the speak method and the
relevant implement will be called - that way if we ever need to add a new type or change one
it doesn't affect the others.
The processing itself is done inside the DirectoryProcessor class - there we print the relevant
warnings if needed and then filter, sort and print the files.

==========
= Answers to questions =
==========
1. Describe the exceptions hierarchy you used in order to handle errors in the program. Explain
the considerations that made you choose that specific design.
If the error was related to something specific to this project like the source folder is empty
or the Commands file is not valid then I created a custom Exceptions inside the package
in DirectoryProcessingExceptions.java.
I raised up to the main class if we needed to end the processing (meaning type 2 error)
otherwise if it was a type-1 error I only saved the warning message for future export
and dealt with the exception inside the class.
I used also common exceptions such as NullPointerException and UnsupportedOperationException
If for example something went wrong with retrieving the files' info or processing
the regex expressions for the Commands file.
3. How did you sort your matched files? Did you use a data structure for this purpose? If so,
what data structure and why?
For the sorting itself I used a quicksort algorithm. I used an ArrayList
that way I didn't need to know in advance to number of operations it will need to store.
Also, that way I could easily do the reverse operation thanks to Collections.reverse.
Another point I considered was that the order of insert is preserved.

3 filesprocessing/DirectoryProcessingExceptions.java

```
1  package filesprocessing;
2
3  /**
4   * Class for exceptions related to the processing.
5   * @author Bar Melinarskiy
6   * @version 8/9/20
7   */
8  public class DirectoryProcessingExceptions
9  {
10     /**
11      * Nested exception class for a bad format of Commands File.
12      * @author Bar Melinarskiy
13      * @version 8/9/20
14      */
15     protected static class BadCommandsFileException extends Exception
16     {
17         /**
18          * Create a new BadCommandsFileException exception
19          * @param errorMessage the error message to throw
20          */
21         public BadCommandsFileException(String errorMessage) {
22             super(errorMessage);
23         }
24     }
25     /**
26      * Nested exception class for an empty commands file error.
27      * @author Bar Melinarskiy
28      * @version 8/9/20
29      */
30     protected static class EmptyFileException extends Exception
31     {
32         /**
33          * Create a new EmptyFileException exception
34          * @param errorMessage the error message to throw
35          */
36         public EmptyFileException(String errorMessage) {
37             super(errorMessage);
38         }
39     }
40     /**
41      * Nested exception class for an empty source foldr.
42      * @author Bar Melinarskiy
43      * @version 8/9/20
44      */
45     protected static class EmptyDirectory extends Exception
46     {
47         /**
48          * Create a new EmptyDirectory exception
49          * @param errorMessage the error message to throw
50          */
51         public EmptyDirectory(String errorMessage) {
52             super(errorMessage);
53         }
54     }
55 }
```

4 filesprocessing/DirectoryProcessor.java

```java
package filesprocessing;
import java.util.ArrayList;
/**
 * The main class running the files processing according to user's input.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class DirectoryProcessor
{
    // constants
    private static final int ERROR_EXIT_CODE = -1;
    // instance variables
    private static ArrayList<FileInfo> filesInfo = new ArrayList<FileInfo>();
    private static ArrayList<ProcessOperation> processOperations = new ArrayList<ProcessOperation>();;
    /**
     * Creates a new game.
     *
     * @param args the command line arguments.
     * @throws DirectoryProcessingExceptions.EmptyDirectory
     * @throws DirectoryProcessingExceptions.BadCommandsFileException
     * @throws DirectoryProcessingExceptions.EmptyFileException
     * @throws NullPointerException
     * @throws UnsupportedOperationException
     */
    public DirectoryProcessor(String[] args)
        throws DirectoryProcessingExceptions.EmptyDirectory,
               DirectoryProcessingExceptions.BadCommandsFileException,
               DirectoryProcessingExceptions.EmptyFileException,
               NullPointerException,
               UnsupportedOperationException
    {
        final int MIN_ARGS_COUNT = 2;
        final int SOURCE_DIRECTORY_INDEX = 0;
        final int COMMANDS_INDEX = 1;
        if(args.length == MIN_ARGS_COUNT)
        {
            String sourceDirectoryPath = args[SOURCE_DIRECTORY_INDEX];
            String commandsFilePath = args[COMMANDS_INDEX];
            filesInfo = FileInfoFactory.createFilesInfoArray(sourceDirectoryPath);
            processOperations = ProcessOperationFactory.createProcessingOperations(commandsFilePath);
        }
        else
        {
            printUsageAndExit();
        }
    }
    /**
     * Prints a usage message and exit.
     * @throws UnsupportedOperationException
     */
    private static void printUsageAndExit()
        throws UnsupportedOperationException
    {
        throw new UnsupportedOperationException("ERROR: usage error, you must enter 2 valid paths");
    }
    /**
     * Runs the files processing.
     */
    private void process()
```

```
60      {
61          for(ProcessOperation process : processOperations)
62          {
63              process.run(filesInfo);
64          }
65      }
66
67      /**
68       * main function, get the arguments from the user & Runs the files processing.
69       *
70       * @param args command line arguments.
71       */
72      public static void main(String[] args)
73      {
74          try
75          {
76              DirectoryProcessor filesProcessing = new DirectoryProcessor(args);
77              filesProcessing.process();
78          }
79          catch(DirectoryProcessingExceptions.EmptyDirectory | UnsupportedOperationException |
80                  DirectoryProcessingExceptions.BadCommandsFileException e)
81          {
82              System.err.println(e.getMessage());
83              return;
84          }
85          catch(DirectoryProcessingExceptions.EmptyFileException | NullPointerException e)
86          {
87              return;
88          }
89      }
90  }
```

5 filesprocessing/FileInfo.java

```java
package filesprocessing;
import java.io.File;
/**
 * Class to store each file needed info for future usage.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FileInfo
{
    // constants
    private static final double INIT_SIZE = 0;
    // instance variables
    private double size = INIT_SIZE;
    private String absName;
    private String name;
    private String type;
    private Boolean isHidden = false;
    private Boolean isWritable = false;
    private Boolean isExecutable = false;
    /*----=  Constructor  =-----*/
    /**
     * Construct a file info object from given filepath.
     * @param file file to fetch info from.
     * @throws NullPointerException if the specified file is null
     */
    public FileInfo(File file)
    {
        try
        {
            if(file != null)
            {
                name = FileUtils.getName(file);
                absName = FileUtils.getAbsName(file);
                type = FileUtils.getType(file);
                size = FileUtils.getFileSizeKiloBytes(file);
                isHidden = FileUtils.checkIfHidden(file);
                isWritable = FileUtils.checkIfWritable(file);
                isExecutable = FileUtils.checkIfExecutable(file);
            }
            else
            {
                throwErrorInConstructor();
            }
        }
        catch (Exception e)
        {
            throwErrorInConstructor();
        }
    }
    // instance methods
    /**
     * Get the file's name.
     * @return the file's name.
     */
    public String getName()
    {
        return name;
    }

```

```
60      /**
61       * Get the file's abs name.
62       * @return the file's abs name.
63       */
64      public String getAbsName()
65      {
66          return absName;
67      }
68
69      /**
70       * Get the file's type.
71       * @return the file's name.
72       */
73      public String getType()
74      {
75          return type;
76      }
77      /**
78       * Get the executable flag.
79       * @return The executable flag, true if it is indeed executable
80       * false otherwise.
81       */
82      public Boolean getExecutable()
83      {
84          return isExecutable;
85      }
86      /**
87       * Get the hidden flag.
88       * @return The hidden flag, true if it is indeed hidden
89       * false otherwise.
90       */
91      public Boolean getHidden()
92      {
93          return isHidden;
94      }
95      /**
96       * Get the writable flag.
97       * @return The writable flag, true if it is indeed writable
98       * false otherwise.
99       */
100     public Boolean getWritable()
101     {
102         return isWritable;
103     }
104     /**
105      * Get the file's size.
106      * @return The file's size.
107      */
108     public double getSize()
109     {
110         return size;
111     }
112     /**
113      * throw an error in the Constructor.
114      * @throws NullPointerException if the specified file is null
115      */
116     private void throwErrorInConstructor()
117     {
118         System.err.println("ERROR: Could not retrieve info of file in folder.");
119         throw new NullPointerException();
120     }
121
122 }
```

6 filesprocessing/FileInfoFactory.java

```java
package filesprocessing;
import java.io.File;
import java.util.ArrayList;

/**
 * Class for retrieving the files' info from the source directory.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FileInfoFactory
{
    /**
     * Get an array of all files' info inside the given folder
     * @param sourceDirectoryPath source directory path to check.
     * @return an array of the files inside the source directory
     * @throws DirectoryProcessingExceptions.EmptyDirectory if the source folder is empty
     */
    public static ArrayList<FileInfo> createFilesInfoArray(String sourceDirectoryPath)
            throws DirectoryProcessingExceptions.EmptyDirectory {
        final int NO_FILES = 0;
        File[] files = FileUtils.getAllFilesInFolder(sourceDirectoryPath);
        if(files != null)
        {
            int numberOfFiles = files.length;
            //Check the folder isn't empty
            if(numberOfFiles == NO_FILES)
            {
                throw new DirectoryProcessingExceptions.EmptyDirectory("ERROR: No files in sourcedir");
            }
            //loop on all the files and get the needed info for later usage
            ArrayList<FileInfo> filesInfo = new ArrayList<FileInfo>();
            for (File file : files)
            {
                filesInfo.add(new FileInfo(file));
            }
            return filesInfo;
        }
        return null;
    }
}
```

7 filesprocessing/FileUtils.java

```java
package filesprocessing;
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * A utilities class for file manipulation.
 *
 */
public class FileUtils
{
    /**
     * Reads a text file (such that each line contains a single word),
     * and returns a string array of its lines.
     * @param fileName Text file to read.
     * @return Array with the file's content (returns null if the IOException occurred).
     */
    public static String[] file2array(String fileName)
    {
        // A list to hold the file's content
        List<String> fileContent = new ArrayList<String>();

        // Reader object for reading the file
        BufferedReader reader = null;

        try
        {
            // Open a reader
            reader = new BufferedReader(new FileReader(fileName));

            // Read the first line
            String line = reader.readLine();

            // Go over the rest of the file
            while (line != null)
            {

                // Add the line to the list
                fileContent.add(line);

                // Read the next line
                line = reader.readLine();
            }

        } catch (FileNotFoundException e)
        {
            System.err.println("ERROR: The file: " + fileName + " is not found.");
            return null;
        } catch (IOException e)
        {
            System.err.println("ERROR: An IO error occurred.");
            return null;
        } finally
```

```
60          {
61              // Try to close the file
62              try
63              {
64                  if (reader != null)
65                  {
66                      reader.close();
67                  }
68                  else
69                  {
70                      return null;
71                  }
72
73              }
74              catch (IOException e)
75              {
76                  System.err.println("ERROR: Could not close the file " + fileName + ".");
77              }
78          }
79
80          // Convert the list to an array and return the array
81          String[] result = new String[fileContent.size()];
82          fileContent.toArray(result);
83          return result;
84      }
85      /**
86       * Get the file size
87       * @param file file to check.
88       * @return file size.
89       */
90      protected static double getFileSizeKiloBytes (File file)
91      {
92          final int CONVERT_TO_KB = 1024;
93          return (double) file.length() / CONVERT_TO_KB;
94      }
95      /**
96       * Check if the given file is hidden or not
97       * @param file file to check.
98       * @return true if the file is indeed hidden, false otherwise.
99       */
100     protected static Boolean checkIfHidden(File file)
101     {
102         if (file != null && (file.isHidden() || file.getName().startsWith(".")))
103         {
104             return true;
105         }
106         return false;
107     }
108
109     /**
110      * Check if the given file is executable or not
111      * @param file file to check.
112      * @return true if the file is indeed executable, false otherwise.
113      */
114     protected static Boolean checkIfExecutable(File file)
115     {
116         if (file != null && file.canExecute())
117         {
118             return true;
119         }
120         return false;
121     }
122
123     /**
124      * Check if the given file is writable or not
125      * @param file file to check.
126      * @return true if the file is indeed writable, false otherwise.
127      */
```

```
128     protected static Boolean checkIfWritable(File file)
129     {
130         if (file != null && file.canWrite())
131         {
132             return true;
133         }
134         return false;
135     }
136 
137     /**
138      * Get the given file name
139      * @param file file to check.
140      * @return file's name.
141      */
142     protected static String getName(File file)
143     {
144         return file.getName();
145     }
146 
147     /**
148      * Get the given file abs name
149      * @param file file to check.
150      * @return file's abs name.
151      */
152     protected static String getAbsName(File file)
153     {
154         return file.getAbsolutePath();
155     }
156     /**
157      * Get a list of all files inside the given folder
158      * @param folderPath folder to check.
159      * @return list of files inside the given folder.
160      */
161     protected static File[] getAllFilesInFolder(String folderPath)
162     {
163         // try-catch block to handle exceptions
164         try
165         {
166 
167             // Create a file object
168             File directory = new File(folderPath);
169 
170             //First we create a FileFilter, get only files and no directories
171             FileFilter filter = new FileFilter()
172             {
173                 @Override
174                 public boolean accept(File f)
175                 {
176                     return Files.isRegularFile(f.toPath());
177                 }
178             };
179             // Get all  the files present in the given directory
180             File[] files = directory.listFiles(filter);
181             if(files != null)
182             {
183                 return files;
184             }
185         }
186         catch (Exception e)
187         {
188             System.err.println("ERROR: Could not retrieve files from folder " + folderPath + ".");
189         }
190         //if we reached this point then we didn't managed to get the files from the folder
191         System.err.println("ERROR: Could not retrieve files from folder " + folderPath + ".");
192         return null;
193     }
194 
195     /**
```

```
196         * Get the given file type
197         * @param file file to check.
198         * @return file's type.
199         */
200        public static String getType(File file)
201        {
202            String type = "";
203            String name = file.getName();
204            int i = name.lastIndexOf('.');
205            if (i > 0)
206            {
207                type = name.substring(i + 1);
208            }
209            return type;
210        }
211    }
```

8 filesprocessing/ProcessOperation.java

```java
package filesprocessing;

import java.util.ArrayList;

/**
 * Class to store processing actions from the commands file.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class ProcessOperation extends ProcessOperationCore
{
    //consts
    private final static int INIT_SIZE = 0;
    /**
     * Run the process operation
     * @param filesInfo the files inside the source folder to process
     */
    protected void run(ArrayList<FileInfo> filesInfo)
    {
        issueWarningIfNeeded();
        ArrayList<FileInfo> filteredFiles = runFilter(filesInfo);
        if(filteredFiles.size() > INIT_SIZE)
        {
            runSort(filteredFiles);
            printFiles(filteredFiles);
        }
    }
    /**
     * Print the files
     * @param files the files to print
     */
    private void printFiles(ArrayList<FileInfo> files)
    {
        for(FileInfo file : files)
        {
            System.out.println(file.getName());
        }
    }

    /**
     * Run the filter operation
     * @param filesInfo the files inside the source folder to filter
     */
    private ArrayList<FileInfo> runFilter(ArrayList<FileInfo> filesInfo)
    {
        return getFilter().filter(this, filesInfo);
    }

    /**
     * Run the sort operation
     * @param filesInfo the files to sort
     */
    private void runSort(ArrayList<FileInfo> filesInfo)
    {
        getOrder().order(this, filesInfo);
    }
}
```

9 filesprocessing/ProcessOperationCore.java

```java
package filesprocessing;

import filesprocessing.filters.FilesFilter;
import filesprocessing.filters.FilesFilterFactory;
import filesprocessing.filters.FilterCommand;
import filesprocessing.orders.FilesOrder;
import filesprocessing.orders.FilesOrderFactory;
import filesprocessing.orders.OrderCommand;

import java.util.ArrayList;

/**
 * Class to store processing actions from the commands file.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public abstract class ProcessOperationCore
{
    // instance variables
    /** Describes the operation's order command type.
     */
    private FilesOrder order;
    /** Describes the operation's filter command type.
     */
    private FilesFilter filter;
    /** True if the #NOT flag was received.
     */
    private Boolean notFlag = false;
    /** True if the #REVERSE flag was received.
     */
    private Boolean reverseFlag = false;
    /** True if the #YES value was received.
     */
    private Boolean metaFilterFlag = false;
    /** Size related values from the commands file.
     */
    private ArrayList<Double> sizeFilters = new ArrayList<Double>();
    /** File's name related value from the commands file.
     */
    private String nameFilter;
    /** Warning messages to print before running this process.
     */
    private ArrayList<String> warnings = new ArrayList<String>();
    /*----=  Constructor  =-----*/
    /**
     * construct a default new process operation
     * sets the filter to all and the order to abc.
     */
    public ProcessOperationCore()
    {
        order = FilesOrderFactory.createOrder(OrderCommand.ABS);
        filter = FilesFilterFactory.createFilter(FilterCommand.ALL);
    }
    /**
     * Set the process' order command
     * @param orderCommand the new order command
     */
    public void setOrder(OrderCommand orderCommand)
    {
```

```
60          order =  FilesOrderFactory.createOrder(orderCommand);
61      }
62
63      /**
64       * Set the process' filter command
65       * @param filterCommand the new filter command
66       */
67      public void setFilter(FilterCommand filterCommand)
68      {
69          filter = FilesFilterFactory.createFilter(filterCommand);
70      }
71
72      /**
73       * Add to the process warning messages
74       * @param msg the warning message to add
75       */
76      public void addWarning(String msg)
77      {
78          warnings.add(msg);
79      }
80
81      /**
82       * Set the process' size filters values
83       * @param sizeFilters the new size filters to set
84       */
85      public void setSizeFilters(ArrayList<Double> sizeFilters)
86      {
87          this.sizeFilters = sizeFilters;
88      }
89
90      /**
91       * Set the process' name filter value
92       * @param nameFilter the new name filter to set
93       */
94      public void setNameFilter(String nameFilter)
95      {
96          this.nameFilter = nameFilter;
97      }
98
99      /**
100      * Set the process' filter not flag
101      * @param value the new value
102      */
103     public void setNotFlag(Boolean value)
104     {
105         notFlag = value;
106     }
107
108     /**
109      * Set the metadata filter flag for Hidden/Writable/Executable filters
110      * @param value the new value
111      */
112     public void setMetaFilterFlag(Boolean value)
113     {
114         metaFilterFlag = value;
115     }
116
117     /**
118      * Set the process' order reverse flag
119      * @param value the new value
120      */
121     public void setReverseFlag(Boolean value)
122     {
123         reverseFlag = value;
124     }
125
126     /**
127      * Get the process' order command
```

```
128         * @return the order command
129         */
130        public FilesOrder getOrder()
131        {
132            return order;
133        }
134
135        /**
136         * Get the process' filter command
137         * @return the filter command
138         */
139        public FilesFilter getFilter()
140        {
141            return filter;
142        }
143
144        /**
145         * Get the process' filter not flag
146         * @return the process' filter not flag
147         */
148        public Boolean getNotFlag()
149        {
150            return notFlag;
151        }
152
153        /**
154         * Get the process' order reverse flag
155         * @return the process' order reverse flag
156         */
157        public Boolean getReverseFlag()
158        {
159            return reverseFlag;
160        }
161
162        /**
163         * Get the process' size filters values
164         * @return the process' size filters values
165         */
166        public ArrayList<Double> getSizeFilters()
167        {
168            return sizeFilters;
169        }
170
171        /**
172         * Get the process' name filter value
173         * @return the process' name filter value
174         */
175        public String getNameFilter()
176        {
177            return nameFilter;
178        }
179
180        /**
181         * Get the metadata filter flag for Hidden/Writable/Executable filters
182         * @return the metadata filter flag for Hidden/Writable/Executable filters
183         */
184        public Boolean getMetaFilterFlag()
185        {
186            return metaFilterFlag;
187        }
188
189        /**
190         * Get the process warning messages
191         * @return the warning messages
192         */
193        protected ArrayList<String> getWarnings()
194        {
195            return warnings;
```

```
196         }
197
198         /**
199          * Print warnings if needed
200          * @return the warning message
201          */
202         protected void issueWarningIfNeeded()
203         {
204             for(String msg : getWarnings())
205             {
206                 System.err.println(msg);
207             }
208         }
209     }
```

10 filesprocessing/ProcessOperationFactory.java

```java
package filesprocessing;
import filesprocessing.filters.FilterCommand;
import filesprocessing.orders.OrderCommand;

import java.util.ArrayList;
import java.util.MissingFormatArgumentException;

/**
 * Class for retrieving the desired processing tasks from the commands file.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class ProcessOperationFactory
{
    private static boolean isSectionTitleNow = true;
    // constants
    static private final String BAD_COMMANDS_FORMAT = "ERROR: Bad format of Commands File";
    static private final String WARNING_IN_LINE = "Warning in line ";
    /**
     * Get an array of all files' info inside the given folder
     * @param commandsFilePath command file path to check.
     * @return an array of the processing operations
     * @throws DirectoryProcessingExceptions.EmptyFileException if the commands file is empty
     * @throws DirectoryProcessingExceptions.BadCommandsFileException if the commands file
     * is not in the right format
     */
    public static ArrayList<ProcessOperation> createProcessingOperations(String commandsFilePath)
            throws DirectoryProcessingExceptions.EmptyFileException,
                   DirectoryProcessingExceptions.BadCommandsFileException
    {
        String[] lines = FileUtils.file2array(commandsFilePath);
        if(lines != null)
        {
            //Check if the file is empty
            checkIfCommandsEmpty(lines);
            return processLines(lines);
        }
        throw new DirectoryProcessingExceptions.BadCommandsFileException(BAD_COMMANDS_FORMAT);
    }
    /**
     * Process the given lines into an array of process operations
     * @param lines lines from commands file to process.
     * @return an array of the processing operations
     * @throws DirectoryProcessingExceptions.BadCommandsFileException if the commands file is not valid
     */
    private static ArrayList<ProcessOperation> processLines(String[] lines)
            throws DirectoryProcessingExceptions.BadCommandsFileException
    {
        ArrayList<ProcessOperation> processOperations = new ArrayList<ProcessOperation>();
        int numberOfLines = lines.length;
        boolean isFilterSectionNow = true;
        isSectionTitleNow = true;
        for(int i = 0; i < numberOfLines; i++)
        {
            if(isSectionTitleNow)
            {
                isSectionTitleNow = false;
                checkSectionTitle(lines[i], isFilterSectionNow);
            }
```

```
60              else
61              {
62                  isSectionTitleNow = true;
63                  checkSectionContent(lines[i], i, isFilterSectionNow, processOperations);
64                  isFilterSectionNow = !isFilterSectionNow;
65
66              }
67          }
68
69          if(!isSectionTitleNow && isFilterSectionNow)
70          {
71              throw new DirectoryProcessingExceptions.BadCommandsFileException(BAD_COMMANDS_FORMAT);
72          }
73
74          return processOperations;
75      }
76      /**
77       * Check that the file is not empty and have enough lines
78       * @param lines lines from file to check.
79       * @throws DirectoryProcessingExceptions.EmptyFileException if the
80       * section title is not valid
81       * @throws DirectoryProcessingExceptions.BadCommandsFileException if there isn't
82       * enough lines in the file
83       */
84      private static void checkIfCommandsEmpty(String[] lines)
85              throws DirectoryProcessingExceptions.EmptyFileException,
86                     DirectoryProcessingExceptions.BadCommandsFileException
87      {
88          final int NOT_LINES = 0;
89          final int NOT_ENOUGH_LINES = 2;
90          int numberOfLines = lines.length;
91          //Check if the file is empty
92          if(numberOfLines == NOT_LINES)
93          {
94              throw new DirectoryProcessingExceptions.EmptyFileException("Command file is empty");
95          }
96          else if(numberOfLines <= NOT_ENOUGH_LINES)
97          {
98              throw new DirectoryProcessingExceptions.BadCommandsFileException(BAD_COMMANDS_FORMAT);
99          }
100     }
101
102     /**
103      * Check that the order/filter section titles are valid
104      * @param line line to check.
105      * @param isFilterSectionNow flag to indicate whether we are now expecting
106      * a Filter section or not
107      * @throws DirectoryProcessingExceptions.BadCommandsFileException if the
108      * section title is not valid
109      */
110     private static void checkSectionTitle(String line, boolean isFilterSectionNow)
111             throws DirectoryProcessingExceptions.BadCommandsFileException
112     {
113         boolean isOK;
114         if(isFilterSectionNow)
115         {
116             isOK = checkFilterTitle(line);
117         }
118         else
119         {
120             isOK = checkOrderTitle(line);
121         }
122
123         if(!isOK)
124         {
125             throw new DirectoryProcessingExceptions.BadCommandsFileException(BAD_COMMANDS_FORMAT);
126         }
127     }
```

```
128     /**
129      * Check that the order/filter section content is valid
130      * @param line line to check.
131      * @param i index of line.
132      * @param isFilterSectionNow flag to indicate whether we are now expecting
133      * a Filter section or not
134      * @param processOperations array or process operations to fill.
135      */
136     private static void checkSectionContent(String line, int i, boolean isFilterSectionNow,
137                                             ArrayList<ProcessOperation> processOperations)
138     {
139         if(isFilterSectionNow)
140         {
141             processOperations.add(new ProcessOperation());
142             extractFilerInfo(line, i, processOperations);
143         }
144         else
145         {
146             extractOrderInfo(line, i, processOperations);
147         }
148     }
149     /**
150      * Check the order section title is valid
151      * @param line line to check.
152      * @return true if this line is indeed a valid title of the Order section,
153      * false otherwise
154      */
155     private static Boolean checkOrderTitle(String line)
156      {
157         final String orderTitle = "ORDER";
158         return line.equals(orderTitle);
159     }
160     /**
161      * Check the filter section title is valid
162      * @param line line to check.
163      * @return true if this line is indeed a valid title of the Filter section,
164      * false otherwise
165      */
166     private static Boolean checkFilterTitle(String line)
167     {
168         final String filterTitle = "FILTER";
169         return line.equals(filterTitle);
170     }
171
172     /**
173      * Validate the given order line & fetch the Order type and info.
174      * @param line line to check.
175      * @param i index of line.
176      * @param processOperations process operations to fill.
177      */
178     private static void extractOrderInfo(String line, int i, ArrayList<ProcessOperation> processOperations)
179     {
180         final int index = processOperations.size() - 1;
181         //loop through all the valid commands to check this line
182         for (OrderCommand regexOrderCommand : OrderCommand.values())
183         {
184             if(RegexUtils.test(regexOrderCommand.toString(), line))
185             {
186                 processOperations.get(index).setOrder(regexOrderCommand);
187                 extractOrderReverseFlag(line, processOperations);
188                 return;
189             }
190         }
191         //If we reached this point than this line maybe not a valid order command
192         //check if this is new Filter section
193         if(!checkFilterTitle(line))
194         {
195             //This is indeed a non-valid order command show issue a waring
```

```
196             processOperations.get(index).addWarning(WARNING_IN_LINE + (i + 1));
197         }
198         else
199         {
200             isSectionTitleNow = false;
201         }
202         //set the order command
203         processOperations.get(index).setOrder(OrderCommand.ABS);
204     }
205     /**
206      * Check if the order command was with the #REVERSE flag or not
207      * @param line line to check.
208      * @param processOperations process operations to fill.
209      */
210     private static void extractOrderReverseFlag(String line, ArrayList<ProcessOperation> processOperations)
211     {
212         final int index = processOperations.size() - 1;
213         Boolean isReverse = RegexUtils.test(OrderCommand.Constants.SUFFIX_REVERSE, line);
214         processOperations.get(index).setReverseFlag(isReverse);
215     }
216     /**
217      * Validate the given filter line & fetch the Filter type and info.
218      * @param line line to check.
219      * @param i index of line.
220      * @param processOperations process operations to fill.
221      */
222     private static void extractFilerInfo(String line, int i, ArrayList<ProcessOperation> processOperations)
223     {
224         try
225         {
226             final int index = processOperations.size() - 1;
227             //loop through all the valid commands to check this line
228             for (FilterCommand filterCommand : FilterCommand.values())
229             {
230                 if(RegexUtils.test(filterCommand.toString(), line))
231                 {
232                     processOperations.get(index).setFilter(filterCommand);
233                     extractFilterValues(filterCommand, line, processOperations);
234                     extractFilterNotFlag(line, processOperations);
235                     return;
236                 }
237             }
238 
239             setDefaultFiler(i, processOperations);
240         }
241         catch(MissingFormatArgumentException | NullPointerException | NumberFormatException e)
242         {
243             setDefaultFiler(i, processOperations);
244         }
245     }
246     /**
247      * Set the default filter properties after an error was found.
248      * @param i index of line.
249      * @param processOperations process operations to fill.
250      */
251     private static void setDefaultFiler(int i, ArrayList<ProcessOperation> processOperations)
252     {
253         final int index = processOperations.size() - 1;
254         //If we reached this point than this is indeed a non-valid order command so issue a waring
255         processOperations.get(index).addWarning(WARNING_IN_LINE + (i + 1));
256         //set the order filer
257         processOperations.get(index).setFilter(FilterCommand.ALL);
258     }
259 
260     /**
261      * Get the filter value/s for future usage.
262      * @param filterCommand the current filter command from the file.
263      * @param line line to check.
```

```
264         * @param processOperations process operations to fill.
265         * @throws MissingFormatArgumentException if we couldn't get the filter values
266         * @throws NullPointerException  if the string is null
267         * @throws NumberFormatException if the string does not contain
268         * a parsable {@code double}.
269         */
270        private static void extractFilterValues(FilterCommand filterCommand, String line,
271                                                ArrayList<ProcessOperation> processOperations)
272            throws MissingFormatArgumentException,
273                   NullPointerException,
274                   NumberFormatException
275        {
276            final int index = processOperations.size() - 1;
277            final int NUM_OF_VALUES_GT_LT = 1;
278            final int NUM_OF_VALUES_BTW = 2;
279            switch (filterCommand)
280            {
281            case GREATER:
282            case SMALLER:
283                processOperations.get(index).setSizeFilters(RegexUtils.getNumbers(line, NUM_OF_VALUES_GT_LT));
284                break;
285            case BETWEEN:
286                processOperations.get(index).setSizeFilters(RegexUtils.getNumbers(line, NUM_OF_VALUES_BTW));
287                checkRangeValid(processOperations.get(index).getSizeFilters());
288            case CONTAINS:
289            case FILE:
290            case PREFIX:
291            case SUFFIX:
292                processOperations.get(index).setNameFilter(
293                        RegexUtils.getValue(FilterCommand.Constants.VALID_NAME, line));
294                break;
295            case HIDDEN:
296            case WRITABLE:
297            case EXECUTABLE:
298                String bool = RegexUtils.getValue(FilterCommand.Constants.BOOLEAN_VALUE, line);
299                processOperations.get(index).setMetaFilterFlag(bool.equals(FilterCommand.Constants.BOOLEAN_YES));
300            }
301        }
302        /**
303         * Check BTW filter values are valid
304         * @param sizeFilters the array with the given values.
305         * @throws MissingFormatArgumentException if the values are not valid
306         */
307        private static void checkRangeValid(ArrayList<Double> sizeFilters)
308                throws MissingFormatArgumentException
309        {
310            final int INDEX_1 = 0;
311            final int INDEX_2 = 1;
312            double size1 = sizeFilters.get(INDEX_1);
313            double size2 = sizeFilters.get(INDEX_2);
314            if(size2 < size1)
315            {
316                throw new MissingFormatArgumentException("Warning: BTW values range is not valid.");
317            }
318        }
319
320        /**
321         * Check if the filter command was with the #Not flag or not
322         * @param line line to check.
323         * @param processOperations process operations to fill.
324         */
325        private static void extractFilterNotFlag(String line, ArrayList<ProcessOperation> processOperations)
326        {
327            final int index = processOperations.size() - 1;
328            Boolean isNot = RegexUtils.contains(FilterCommand.Constants.NOT_EXIST, line);
329            processOperations.get(index).setNotFlag(isNot);
330        }
331    }
```

11 filesprocessing/RegexUtils.java

```java
package filesprocessing;
import filesprocessing.filters.FilterCommand;

import java.util.ArrayList;
import java.util.MissingFormatArgumentException;
import java.util.regex.*;
/**
 * Utils class for regex related methods.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class RegexUtils
{
    /**
     * Check if given string matches the given regex pattern.
     * @param patternString the regex pattern to test with
     * @param text the string to test on
     * @return true if it was a match, false otherwise
     */
    protected static Boolean test(String patternString, String text)
    {
        Pattern pattern = Pattern.compile(patternString);
        Matcher matcher = pattern.matcher(text);
        return matcher.matches();
    }

    /**
     * Check if given string matches the given regex pattern.
     * @param patternString the regex pattern to test with
     * @param text the string to test on
     * @return true if it was a match, false otherwise
     */
    protected static Boolean contains(String patternString, String text)
    {
        Pattern pattern = Pattern.compile(patternString);
        Matcher matcher = pattern.matcher(text);
        return matcher.find();
    }
    /**
     * Get all the number values from the line.
     * @param text the string to fetch values from
     * @param count the expected count of numbers to be found
     * @return the found values if we found them all, null otherwise
     * @throws MissingFormatArgumentException if we couldn't get the filter values
     * @throws NullPointerException  if the string is null
     * @throws NumberFormatException if the string does not contain
     *         a parsable {@code double}.
     */
    protected static ArrayList<Double> getNumbers(String text, int count)
        throws MissingFormatArgumentException,
               NullPointerException,
               NumberFormatException
    {
        ArrayList<Double> values = new ArrayList<Double>();
        Pattern pattern = Pattern.compile(FilterCommand.Constants.DOUBLE_NUM);
        Matcher matcher = pattern.matcher(text);
        while(matcher.find())
        {
            values.add(Double.parseDouble(matcher.group()));
```

```
60          }
61
62          if(values.size() == count)
63          {
64              return values;
65          }
66          throw new MissingFormatArgumentException("ERROR: Could not retrieve filter values.");
67      }
68      /**
69       * Get the filter value from the line.
70       * @param patternString the regex pattern to test with
71       * @param text the string to fetch value from
72       * @return the found value, null otherwise
73       * @throws MissingFormatArgumentException if we couldn't get the filter values
74       */
75      protected static String getValue(String patternString, String text)
76          throws MissingFormatArgumentException
77      {
78          Pattern pattern = Pattern.compile(patternString);
79          Matcher matcher = pattern.matcher(text);
80          if(matcher.find())
81          {
82              String result =  matcher.group(1);
83              if(result == null)
84              {
85                  result = "";
86              }
87              return result;
88          }
89
90          throw new MissingFormatArgumentException("ERROR: Could not retrieve filter values.");
91      }
92  }
```

12 filesprocessing/filters/FilesFilter.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;
import java.util.ArrayList;
import java.util.stream.Collectors;
/**
 * interface for filtering the files array by some properties
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public interface FilesFilter
{
    /**
     * Filter the files by some property
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
     ArrayList<FileInfo> filter(final ProcessOperation processOperation, final ArrayList<FileInfo> files);

    /**
     * Execute the #NOT command on the filter
     * @param processOperation the current process operation
     * @param files files to filter
     * @param filteredFiles files after filter
     * @return the files who didn't match the filter
     */
    default ArrayList<FileInfo> not(final ProcessOperation processOperation,
                                    final ArrayList<FileInfo> files,
                                    final ArrayList<FileInfo> filteredFiles)
    {
        if(processOperation.getNotFlag())
        {
            return files.stream()
                    .filter(file -> !filteredFiles.contains(file))
                    .collect(Collectors.toCollection(ArrayList::new));
        }
        return filteredFiles;
    }
}
```

13 filesprocessing/filters/FilesFilterFactory.java

```java
package filesprocessing.filters;

/**
 * Class for creating a new filter object fot the file processing operation
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilesFilterFactory
{
    /**
     * Create filter type for process operation
     * @param filterType the current type to create
     */
    public static FilesFilter createFilter(FilterCommand filterType)
    {
        switch(filterType)
        {
        case FILE:
            return new FilterByName();
        case SUFFIX:
            return new FilterBySuffix();
        case PREFIX:
            return new FilterByPrefix();
        case CONTAINS:
            return new FilterByContains();
        case GREATER:
            return new FilterBySizeGT();
        case SMALLER:
            return new FilterBySizeLT();
        case BETWEEN:
            return new FilterBySizeBTW();
        case HIDDEN:
            return new FilterByHidden();
        case EXECUTABLE:
            return new FilterByExecutable();
        case WRITABLE:
            return new FilterByWriteable();
        case ALL:
        default:
            return new FilterAll();
        }
    }
}
```

14 filesprocessing/filters/FilterAll.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Get all the files, no filter.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterAll implements FilesFilter
{
    /**
     * Get all the files
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        return not(processOperation, files, files);
    }
}
```

15 filesprocessing/filters/FilterByContains.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by containing a value their name.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterByContains implements FilesFilter
{
    /**
     * Filter the files containing a value in their name
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        String value = processOperation.getNameFilter();
        ArrayList<FileInfo> filteredFiles =  files.stream()
                .filter(file -> file.getName().contains(value))
                .collect(Collectors.toCollection(ArrayList::new));

        return not(processOperation, files, filteredFiles);
    }
}
```

16 filesprocessing/filters/FilterByExecutable.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by executable property.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterByExecutable implements FilesFilter
{
    /**
     * Filter the files by executable property
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        Boolean flag = processOperation.getMetaFilterFlag();
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> file.getExecutable().equals(flag))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

17 filesprocessing/filters/FilterByHidden.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by visibility.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterByHidden implements FilesFilter
{
    /**
     * Filter the files by their visibility
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        Boolean flag = processOperation.getMetaFilterFlag();
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> file.getHidden().equals(flag))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

18 filesprocessing/filters/FilterByName.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;
import java.util.ArrayList;
import java.util.stream.Collectors;
/**
 * Filter files by their name.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterByName implements FilesFilter
{
    /**
     * Filter the files by their name.
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        String name = processOperation.getNameFilter();
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> file.getName().equals(name))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

19 filesprocessing/filters/FilterByPrefix.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;
/**
 * Filter files by the prefix of their name.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterByPrefix implements FilesFilter
{
    /**
     * Filter the files by the prefix of their name
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        String prefix = processOperation.getNameFilter();
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> file.getName().startsWith(prefix))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

20 filesprocessing/filters/FilterBySizeBTW.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by who's size smaller than given value.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterBySizeBTW implements FilesFilter
{
    /**
     * Filter the files who's size smaller than given value
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        final int INDEX_1 = 0;
        final int INDEX_2 = 1;
        double size1 = processOperation.getSizeFilters().get(INDEX_1);
        double size2 = processOperation.getSizeFilters().get(INDEX_2);
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> (file.getSize() >= size1 && file.getSize() <= size2))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

21 filesprocessing/filters/FilterBySizeGT.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by who's size greater than given value.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterBySizeGT implements FilesFilter
{
    /**
     * Filter the files who's size greater than given value.
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        final int INDEX = 0;
        double size = processOperation.getSizeFilters().get(INDEX);
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> (file.getSize() > size))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

22 filesprocessing/filters/FilterBySizeLT.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by who's size smaller than given value.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterBySizeLT implements FilesFilter
{
    /**
     * Filter the files who's size smaller than given value
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        final int INDEX = 0;
        double size = processOperation.getSizeFilters().get(INDEX);
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> (file.getSize() < size))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

23 filesprocessing/filters/FilterBySuffix.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by the suffix of their name.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterBySuffix implements FilesFilter
{
    /**
     * Filter the files the suffix of their name
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        String suffix = processOperation.getNameFilter();
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> file.getName().endsWith(suffix))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

24 filesprocessing/filters/FilterByWriteable.java

```java
package filesprocessing.filters;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.stream.Collectors;

/**
 * Filter files by writable property.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class FilterByWriteable implements FilesFilter
{
    /**
     * Filter the files by writable property
     *
     * @param processOperation the current process operation
     * @param files files to filter
     * @return the filtered files
     */
    @Override
    public ArrayList<FileInfo> filter(final ProcessOperation processOperation,
                                      final ArrayList<FileInfo> files)
    {
        Boolean flag = processOperation.getMetaFilterFlag();
        ArrayList<FileInfo> filteredFiles = files.stream()
                .filter(file -> file.getWritable().equals(flag))
                .collect(Collectors.toCollection(ArrayList::new));
        return not(processOperation, files, filteredFiles);
    }
}
```

25 filesprocessing/filters/FilterCommand.java

```java
package filesprocessing.filters;
/**
 * All the possible commands.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public enum FilterCommand
{
    GREATER("^greater_than#" + Constants.DOUBLE_NUM + Constants.NOT),
    BETWEEN("^between#" + Constants.DOUBLE_NUM + "#" + Constants.DOUBLE_NUM + Constants.NOT),
    SMALLER("^smaller_than#" + Constants.DOUBLE_NUM + Constants.NOT),
    FILE("^file" + Constants.VALID_NAME + Constants.NOT),
    CONTAINS("^contains" + Constants.VALID_NAME + Constants.NOT),
    PREFIX("^prefix" + Constants.VALID_NAME + Constants.NOT),
    SUFFIX("^suffix" + Constants.VALID_NAME + Constants.NOT),
    WRITABLE("^writable" + Constants.BOOLEAN_VALUE + Constants.NOT),
    EXECUTABLE("^executable" + Constants.BOOLEAN_VALUE + Constants.NOT),
    HIDDEN("^hidden" + Constants.BOOLEAN_VALUE + Constants.NOT),
    ALL("^all" + Constants.NOT);

    private final String value;
    /**
     * Create a new property inside the enum.
     * @param value the inner value
     */
    FilterCommand(final String value)
    {
        this.value = value;
    }

    @Override
    public String toString()
    {
        return value;
    }
    /**
     * Nested class of constants values reused amongst the enum values.
     * @author Bar Melinarskiy
     * @version 8/9/20
     */
    public static class Constants
    {
        public static final String DOUBLE_NUM = "\\+?((\\d+([.]\\d*)?)|([.]\\d+))";
        public static final String VALID_NAME = "#([a-zA-z\\d \\/\\.-_]*)#?";
                                                //"#([a-zA-z]|\\d| |\\/|\\.|-|_)*";
        public static final String BOOLEAN_VALUE = "#(YES|NO)#?";
        public static final String BOOLEAN_YES = "YES";
        public static final String BOOLEAN_NO = "NO";
        public static final String NOT = "(#NOT)*";
        public static final String NOT_EXIST = "(#NOT)";
    }
}
```

26 filesprocessing/orders/FilesOrder.java

```java
package filesprocessing.orders;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;
import java.util.ArrayList;
import java.util.Collections;
/**
 * Class for sorting the files array by some properties.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public abstract class FilesOrder
{
    //consts
      protected static final int EQ = 0;
    /**
     * Help function fot quick sort
     * @param files files to sort
     * @param low Starting index
     * @param high Ending index
     * @return the sorted files
     */
    protected int partition(ArrayList<FileInfo> files, int low, int high)
    {
        FileInfo pivot = files.get(high);
        int i = (low - 1); // index of smaller element
        for (int j = low; j < high; j++)
        {
            // If current element is smaller than the pivot
            if (compare(files.get(j), pivot) < EQ)
            {
                i++;

                // swap files[i] and files[j]
                FileInfo temp = files.get(i);
                files.set(i, files.get(j));
                files.set(j, temp);
            }
        }

        // swap files[i+1] and files[high] (or pivot)
        FileInfo temp = files.get(i + 1);
        files.set(i + 1, files.get(high));
        files.set(high, temp);

        return i+1;
    }

    /**
     * Compare btw two files
     * @param file1 first file.
     * @param file2 second file.
     * @return the value {@code 0} if {@code x == y};
     * a value less than {@code 0} if {@code x < y}; and
     * a value greater than {@code 0} if {@code x > y}
     */
    abstract protected int compare(final FileInfo file1, final FileInfo file2);

    /**
```

```
60          * Sort the files by some property
61          * @param files files to sort
62          * @param low Starting index
63          * @param high Ending index
64          */
65         protected void sort(ArrayList<FileInfo> files, int low, int high)
66         {
67             if (low < high)
68             {
69                 /* pi is partitioning index, files[pi] is
70                   now at right place */
71                 int pi = partition(files, low, high);
72 
73                 // Recursively sort elements before
74                 // partition and after partition
75                 sort(files, low, pi - 1);
76                 sort(files, pi + 1, high);
77             }
78         }
79 
80         /**
81          * Sort the files by some property
82          * @param processOperation the current process operation
83          * @param files files to sort
84          */
85         public void order(final ProcessOperation processOperation,
86                           ArrayList<FileInfo> files)
87         {
88             sort(files,0 , files.size() - 1);
89             reverse(processOperation, files);
90         }
91 
92         /**
93          * Execute the #REVERSE command on the sort if needed
94          * @param processOperation the current process operation
95          * @param files files to sort
96          */
97         protected void reverse(final ProcessOperation processOperation,
98                                        final ArrayList<FileInfo> files)
99         {
100            if(processOperation.getReverseFlag())
101            {
102                Collections.reverse(files);
103            }
104        }
105    }
```

27 filesprocessing/orders/FilesOrderFactory.java

```
1  package filesprocessing.orders;
2
3  /**
4   * Class for creating a new order object for the file processing operation.
5   * @author Bar Melinarskiy
6   * @version 8/9/20
7   */
8  public class FilesOrderFactory
9  {
10     /**
11      * Create order type for process operation
12      * @param orderType the current type to create
13      */
14     public static FilesOrder createOrder(OrderCommand orderType)
15     {
16         switch(orderType)
17         {
18         case SIZE:
19             return new OrderBySize();
20         case TYPE:
21             return new OrderByType();
22         case ABS:
23         default:
24             return new OrderByName();
25         }
26     }
27 }
```

28 filesprocessing/orders/OrderByName.java

```java
package filesprocessing.orders;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

/**
 * Class for sorting the files array by name.
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class OrderByName extends FilesOrder
{
    /**
     * Compare btw two files
     * @param file1 first file.
     * @param file2 second file.
     * @return the value {@code 0} if {@code x == y}; a value less than {@code 0} if {@code x < y}; and a
     *         value greater than {@code 0} if {@code x > y}
     */
    @Override
    protected int compare(FileInfo file1, FileInfo file2)
    {
        int result =  file1.getAbsName().compareTo(file2.getAbsName());
        return result;
    }
}
```

29 filesprocessing/orders/OrderBySize.java

```java
package filesprocessing.orders;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

/**
 * Class for sorting the files array by size, if size eq than by name
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class OrderBySize extends FilesOrder
{
    /**
     * Compare btw two files
     * @param file1 first file.
     * @param file2 second file.
     * @return the value {@code 0} if {@code x == y}; a value less than {@code 0} if {@code x < y}; and a
     *         value greater than {@code 0} if {@code x > y}
     */
    @Override
    protected int compare(FileInfo file1, FileInfo file2)
    {
        int sizeCompare = Double.compare(file1.getSize(), file2.getSize());
        int nameCompare = file1.getAbsName().compareTo(file2.getAbsName());

        // 2-level comparison using if-else block
        if (sizeCompare == 0)
        {
            return ((nameCompare == 0) ? sizeCompare : nameCompare);
        } else
        {
            return sizeCompare;
        }
    }
}
```

30 filesprocessing/orders/OrderByType.java

```java
package filesprocessing.orders;

import filesprocessing.FileInfo;
import filesprocessing.ProcessOperation;

import java.util.ArrayList;
import java.util.Collections;
/**
 * Class for sorting the files array by type, if type eq than by name
 * @author Bar Melinarskiy
 * @version 8/9/20
 */
public class OrderByType extends FilesOrder
{
    /**
     * Compare btw two files
     * @param file1 first file.
     * @param file2 second file.
     * @return the value {@code 0} if {@code x == y}; a value less than {@code 0} if {@code x < y}; and a
     *         value greater than {@code 0} if {@code x > y}
     */
    @Override
    protected int compare(FileInfo file1, FileInfo file2)
    {
        int typeCompare = file1.getType().compareTo(file2.getType());
        int nameCompare = file1.getAbsName().compareTo(file2.getAbsName());

        // 2-level comparison using if-else block
        if (typeCompare == 0)
        {
            return ((nameCompare == 0) ? typeCompare : nameCompare);
        } else
        {
            return typeCompare;
        }
    }
}
```

31 filesprocessing/orders/OrderCommand.java

```
1  package filesprocessing.orders;
2  /**
3   * All the possible Orders.
4   * @author Bar Melinarskiy
5   * @version 8/9/20
6   */
7  public enum OrderCommand
8  {
9      ABS("^abs" + Constants.REVERSE),
10     TYPE("^type" + Constants.REVERSE),
11     SIZE("^size" + Constants.REVERSE);
12
13     private final String value;
14     /**
15      * Create a new property inside the enum.
16      * @param value the inner value
17      */
18     OrderCommand(final String value)
19     {
20         this.value = value;
21     }
22
23     @Override
24     public String toString()
25     {
26         return value;
27     }
28
29     /**
30      * Nested class of constants values reused amongst the enum values.
31      * @author Bar Melinarskiy
32      * @version 8/9/20
33      */
34     public static class Constants
35     {
36         public static final String REVERSE = "(#REVERSE)*";
37         public static final String SUFFIX_REVERSE = ".*#REVERSE";
38     }
39 }
```