Contents

1 Basic Test Results

```
1  ******************** OOP pre-submission script for ex5 ********************
2
3  Extracting jar file...
4
5  /tmp/bodek.GbL7tu/oop/ex5/muaz.abdeen/presubmission/testdir/26945
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

```
muaz.abdeen



============================
=      File description     =
============================
## filesprocessing Package ##

= Filter subpackage =
    - FilterFactory - A factory class for filters used in Directory Processor.
    - NegatedFilter - A decorator class for FileFilter.

= Order subpackage =
    - OrderFactory - A factory class for orders used in Directory Processor.

= ProcessorExceptions subpackage =
    - ProcessingExceptions - Super class for Processing Exceptions.
    - TypeIError - Type I errors (Warnings) class.
    - TypeIIError - Type II errors class.

- DirectoryProcessor - A program thar filters the files in a given directory
                       according to various conditions, and orders the filenames
                       that passed the filtering according to various properties.
- FileSort - User-defined sorting class that uses QuickSort algorithm.
- Section - Represents a single section of command file.
- SectionBuilder - Builds the whole sections in the command file.



============================
=          Design          =
============================

(1) Processing Exceptions
    I create a super class ProcessingExceptions that extends the Exception class,
    and have two subclasses, TypeIError and TypeIIError, so that I can (if needed)
    throw a general ProcessingExceptions, and catch any one of his subclasses.
    Also, this ProcessingExceptions class represents a facade class of Exception
    class, it only has the constructor in its API.

(2) Filter
    I designed a factory class that produces all of the filters, following the Single
    Choice Principle. also I made a decorator class that negates any filter passed to it.
    This design gives my code the extendability and continuity, because if in future I
    need to add more filters I can easily add them to the factory class.

(3) Order
    Here too, I followed the Single Choice Principle, I created a factory class for orders.
    Similar to Filter design this gives my code the extendability and continuity.



============================
=  Implementation details   =
============================

(1) Handling exceptions:
    I handled TypeIError while executing each section in the instance method
```

```
    executeCommands(args) of the Section class.
    I handled TypeIIError in the main method of DirectoryProcessor.
    In any other place I just throw, or rethrow the exception.

(2) Filter:
    I took advantage of the Java's Functional Interface FileFilter.
    In the FilterFactory I used the lambda expression to implement the FileFilter
    for all of the filers given in the exercise description.

(3) Order:
    Like Filter, I took advantage of the Java's Comparator interface, which is also
    a Functional Interface, and in the OrderFactory I used the lambda expression to
    implement the Comparator for each order given in the exercise description.

(4) Sorting:
    I implemented I special version of quick sort to sort lists of File objects.
    The constructor of the FileSort class takes a comparator, which will be used
    to compare File objects in the sorting algorithm.

(5) Printing the results:
    For every section in the command file, I instantiated the Section class, this
    instance contains the section details, according to this details every section
    instance creates a list of filtered and order files in the directory, then after
    printing Type I Errors, the files of this list are printed.
    If while building any section, an exception of Type II was thrown, the process
    stops, and the main method prints the error message.

============================
=    Answers to questions   =
============================
Q - How did you sort your matched files? Did you use a data structure for this purpose?
    If so, what data structure and why?
A - I implemented a special instance of quick sort, that sorted a list of File objects.
    I used the java's ArrayList, because this is list my program made after filtering
    the files in the directory, and passed it the sorter.
```

3 filesprocessing/DirectoryProcessor.java

```java
package filesprocessing;

import filesprocessing.ProcessingExceptions.TypeIIError;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

/**
 * A program thar filters the files in a given directory according to various conditions,
 * and orders the filenames that passed the filtering according to various properties.
 */
public class DirectoryProcessor {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    private final static String USAGE_ERROR = "Usage: java filesprocessing.DirectoryProcessor " +
                                              "<sourcedir> <commandfile>";
    private final static String ACCESS_COMMAND_FILE_ERROR = "Can not access the Commands File.";
    private final static String SOURCE_DIRECTORY_ERROR = "Can not access the Source Directory.";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    /** A list of all files in the source directory */
    private File[] rawFiles;


    /** A list of all lines in the command file */
    private List<String> commandFileLines;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Constructs the directory processor.
     * @param sourceDir a directory name, in the form of a path.
     * @param commandFile a name of a file, also in the form of a relative or absolute path.
     * @throws TypeIIError Type II exception of processing
     */
    DirectoryProcessor(String sourceDir, String commandFile) throws TypeIIError {
        try {
            File sourceDirPath = new File(sourceDir);
            File commandFilePath = new File(commandFile);
            if (!sourceDirPath.exists())
                throw new TypeIIError(SOURCE_DIRECTORY_ERROR);
            if (!commandFilePath.exists())
                throw new TypeIIError(ACCESS_COMMAND_FILE_ERROR);
            rawFiles = sourceDirPath.listFiles(File::isFile);
            commandFileLines = Files.readAllLines(Paths.get(commandFile));
        } catch (IOException e) {
            throw new TypeIIError(ACCESS_COMMAND_FILE_ERROR);
        } catch (NullPointerException e) {
```

```
60              throw new TypeIIError(SOURCE_DIRECTORY_ERROR);
61          }
62      }
63
64      // ################## //
65      // #### METHODS #### //
66      // ################## //
67
68      private void execute() throws TypeIIError {
69          SectionBuilder builder = new SectionBuilder(commandFileLines);
70          for (Section section: builder.createSections()) {
71              section.executeCommands(rawFiles);
72          }
73      }
74
75      // ###################### //
76      // #### MAIN METHOD #### //
77      // ###################### //
78
79      /**
80       * Main function that runs the program
81       * @param args command line arguments to run the program
82       */
83      public static void main(String[] args) {
84          try {
85              if (args.length != 2)
86                  throw new TypeIIError(USAGE_ERROR);
87              DirectoryProcessor processor = new DirectoryProcessor(args[0], args[1]);
88              processor.execute();
89          } catch (TypeIIError exception) {
90              System.err.println(exception.getMessage());
91          }
92      }
93  }
```

4 filesprocessing/FileSort.java

```java
package filesprocessing;

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

/**
 * User-defined sorting class that uses QuickSort algorithm
 */
public class FileSort {

    // ###################### //
    // #### ATTRIBUTES #### //
    // ###################### //

    /** The file comparator used in this sorter */
    private Comparator<File> fileComparator;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Constructs a new sorter
     * @param order The file comparator used in this sorter
     */
    FileSort(Comparator<File> order) {
        fileComparator = order;
    }

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * The partition function helps the quick sorts.
     * @param filesList list to be sorted.
     * @param left left most index.
     * @param right right most index.
     * @return the index of the pivot, such that all objects in the left are
     *           smaller than it, and all in the right are greater than it.
     */
    private int partition(ArrayList<File> filesList, int left, int right) {
        int i = left, j = right;
        File tmp;
        File pivot = filesList.get((left + right) / 2);
        while (i <= j) {
            // filesList[i] < pivot
            while (fileComparator.compare(filesList.get(i), pivot) < 0)
                i++;
            // filesList[j] > pivot
            while (fileComparator.compare(filesList.get(j), pivot) > 0)
                j--;
            if (i <= j) {
                // swap
                tmp = filesList.get(i);
                filesList.set(i, filesList.get(j));
                filesList.set(j, tmp);
                i++;
```

```
60                  j--;
61              }
62          }
63          return i;
64      }
65
66      /**
67       * The quick sort function (sorts in ascending order)
68       * @param filesList list to be sorted.
69       * @param left left most index.
70       * @param right right most index.
71       */
72      void quickSort(ArrayList<File> filesList, int left, int right) {
73          int index = partition(filesList, left, right);
74          if (left < index - 1)
75              quickSort(filesList, left, index - 1);
76          if (index < right)
77              quickSort(filesList, index, right);
78      }
79
80  }
81  // End of FileSort Class
```

5 filesprocessing/Section.java

```java
package filesprocessing;

import filesprocessing.Filter.FilterFactory;
import filesprocessing.ProcessingExceptions.TypeIError;
import filesprocessing.Order.OrderFactory;

import java.io.File;
import java.io.FileFilter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;

/**
 * Represents a single section of command file
 */
public class Section {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    /** The delimiter between command parameters */
    private final static String DELIMITER = "#";

    /** The default order: by absolute name, in ascending order */
    private final static String DEFAULT_ORDER = "abs";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    /** The filer command line before parsing */
    private final String filterCommand;

    /** The parsed parts of the filter command */
    private final int filterLine;
    private String filterName;
    private String[] filterValues = new String[0];
    private boolean negatedFilter = false;

    /** The order command line before parsing */
    private final String orderCommand;

    /** The parsed parts of the order command */
    private final int orderLine;
    private String orderName = DEFAULT_ORDER;
    private boolean reversedOrder = false;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Constructs a new section
     * @param filterCmd filter command
     * @param filterLineNum filter command line number
     * @param orderCmd order command
     * @param orderLineNum order command line number
     */
```

```
60      Section(String filterCmd, int filterLineNum,
61              String orderCmd, int orderLineNum) {
62          filterCommand = filterCmd;
63          filterLine = filterLineNum;
64          orderCommand = orderCmd;
65          orderLine = orderLineNum;
66      }
67
68      // ################## //
69      // #### METHODS #### //
70      // ################## //
71
72      /**
73       * Executing the current section commands
74       * @param rawFiles list of all files before processing
75       */
76      void executeCommands(File[] rawFiles) {
77          ArrayList<File> filteredFiles = new ArrayList<>();
78          // filter files array
79          FileFilter sectionFilter = getFilter();
80          for (File pathname: rawFiles) {
81              if (sectionFilter.accept(pathname))
82                  filteredFiles.add(pathname);
83          }
84          // sort files
85          FileSort sorter = new FileSort(getOrder());
86          if (filteredFiles.size() > 0)
87              sorter.quickSort(filteredFiles, 0, filteredFiles.size() - 1);
88          // print files names
89          for (File pathname: filteredFiles)
90              System.out.println(pathname.getName());
91      }
92
93      /**
94       * Gets the suitable file filter
95       * @return the file filter corresponding to the filter command
96       */
97      private FileFilter getFilter() {
98          try {
99              parseFilter(filterCommand, filterLine);
100             return FilterFactory.select(filterName, filterValues, filterLine, negatedFilter);
101         } catch (TypeIError exception) {
102             System.err.println(exception.getMessage());
103             // Default filter (="all")
104             return pathname -> true;
105         }
106     }
107
108     /**
109      * Gets the suitable file order
110      * @return the file order corresponding to the order command
111      */
112     private Comparator<File> getOrder() {
113         try {
114             // if order command is an empty line return default order (i.e. abs)
115             if (orderCommand.equals("") && orderLine == -1)
116                 return Comparator.comparing(File::getAbsolutePath);
117             parseOrder(orderCommand, orderLine);
118             return OrderFactory.select(orderName, orderLine, reversedOrder);
119         } catch (TypeIError exception) {
120             System.err.println(exception.getMessage());
121             // Default order (="abs")
122             return Comparator.comparing(File::getAbsolutePath);
123         }
124     }
125
126     /**
127      * Parsing the filter command.
```

```
128         * @param filterCommand filter command of this section
129         * @param filterLineNum filter command line
130         * @throws TypeIError WARNING
131         */
132        private void parseFilter(String filterCommand, int filterLineNum) throws TypeIError {
133            String[] commandParts = filterCommand.split(DELIMITER);
134            // filter command is composed of 1-4 parts (includes NOT)
135            if (commandParts.length < 1 || commandParts.length > 4) {
136                throw new TypeIError(filterLineNum);
137            }
138            filterName = commandParts[0];
139            if (commandParts.length > 1) {
140                negatedFilter = commandParts[commandParts.length-1].equals("NOT");
141                int argsNum = (negatedFilter) ? commandParts.length-2 : commandParts.length-1;
142                filterValues = Arrays.copyOfRange(commandParts, 1, argsNum+1);
143            }
144        }
145
146        /**
147         * Parsing the order command.
148         * @param orderCommand order command of this section
149         * @param orderLineNum order command line
150         * @throws TypeIError WARNING
151         */
152        private void parseOrder(String orderCommand, int orderLineNum) throws TypeIError {
153            String[] commandParts = orderCommand.split(DELIMITER);
154            // If there is an order command, it is composed of 1-2 parts (includes REVERSE)
155            if (commandParts.length == 1) {
156                orderName = commandParts[0];
157            } else if (commandParts.length == 2) {
158                orderName = commandParts[0];
159                reversedOrder = commandParts[1].equals("REVERSE");
160                // the reverse part (2nd one) not equals to "REVERSE"
161                if (!reversedOrder)
162                    throw new TypeIError(orderLineNum);
163            } else if (commandParts.length > 2) {
164                throw new TypeIError(orderLineNum);
165            }
166        }
167    }
168    // End of Section Class
```

6 filesprocessing/SectionBuilder.java

```java
package filesprocessing;

import filesprocessing.ProcessingExceptions.TypeIIError;

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

/**
 * Builds the whole sections in the command file
 */
public class SectionBuilder {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    final static String FILTER_TITLE = "FILTER";
    private final static String FILTER_MISSING = "FILTER title is missing.";
    final static String ORDER_TITLE = "ORDER";
    private final static String ORDER_MISSING = "ORDER title is missing.";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    /** List of all commands in command file */
    private final List<String> fileCommands;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * Constructs a new section builder.
     * @param commandFileLines List of all commands in command file.
     */
    SectionBuilder(List<String> commandFileLines) {
        fileCommands = commandFileLines;
    }

    // ################# //
    // #### METHODS #### //
    // ################# //

    /**
     * Creates an ArrayList of all sections of the command file.
     * @return ArrayList of all sections of the command file.
     * @throws TypeIIError ERROR Exceptions.
     */
    ArrayList<Section> createSections() throws TypeIIError {
        ArrayList<Section> sections = new ArrayList<>();
        int idx = 0;
        try {
            while (idx < fileCommands.size()) {
                String filterCommand, orderCommand;
                int filterLineNum, orderLineNum;
                // Check FILTER sub-section
                if (!fileCommands.get(idx).equals(FILTER_TITLE))
                    throw new TypeIIError(FILTER_MISSING);
```

```
                filterCommand = fileCommands.get(idx + 1);
                filterLineNum = idx + 2;
                // Checks ORDER sub_section
                if (!fileCommands.get(idx + 2).equals(ORDER_TITLE))
                    throw new TypeIIError(ORDER_MISSING);
                // No command in ORDER sub-section
                if ((idx + 3 >= fileCommands.size()) || (fileCommands.get(idx + 3).equals(FILTER_TITLE))) {
                    orderCommand = "";
                    orderLineNum = -1;
                    idx += 3;  // section is 3 lines long
                } else {
                    orderCommand = fileCommands.get(idx + 3);
                    orderLineNum = idx + 4;
                    idx += 4; // section is 4 lines long
                }
                sections.add(new Section(filterCommand, filterLineNum,
                                         orderCommand, orderLineNum));
            }
        } catch (IndexOutOfBoundsException e) {
            // there is only the FILTER title line in this section
            throw new TypeIIError(FILTER_MISSING);
        }
        return sections;
    }
}
// End of SectionBuilder Class
```

7 filesprocessing/Filter/FilterFactory.java

```java
package filesprocessing.Filter;

import filesprocessing.ProcessingExceptions.TypeIError;

import java.io.FileFilter;

/**
 * A factory class for filters used in Directory Processor
 */
public class FilterFactory {

    private static final double BYTES_IN_KB = 1024;

    /**
     * Returns the selected file filter
     * @param filterName filter name
     * @param filterValues filter parameters
     * @param filterLine line number
     * @param negatedFilter is the filter negated?
     * @return The selected file filter
     * @throws TypeIError WARNINGS
     */
    public static FileFilter select(String filterName, String[] filterValues,
                                    int filterLine, boolean negatedFilter) throws TypeIError {
        FileFilter filter;
        try {
            switch (filterName) {
            case "greater_than":
                filter = greaterThan(filterValues, filterLine);
                break;
            case "between":
                filter = between(filterValues, filterLine);
                break;
            case "smaller_than":
                filter = smallerThan(filterValues, filterLine);
                break;
            case "file":
                filter = file(filterValues, filterLine);
                break;
            case "contains":
                filter = contains(filterValues, filterLine);
                break;
            case "prefix":
                filter = prefix(filterValues, filterLine);
                break;
            case "suffix":
                filter = suffix(filterValues, filterLine);
                break;
            case "writable":
                filter = writable(filterValues, filterLine);
                break;
            case "executable":
                filter = executable(filterValues, filterLine);
                break;
            case "hidden":
                filter = hidden(filterValues, filterLine);
                break;
            case "all":
                filter = all(filterValues, filterLine);
```

```
60                  break;
61              default: // filter name not matching any case
62                  throw new TypeIError(filterLine);
63              }
64              // check if filter is negated
65              return (negatedFilter) ? new NegatedFilter(filter): filter;
66          } catch (TypeIError e) {
67              throw new TypeIError(filterLine);
68          }
69      }
70
71      /**
72       * Tests whether or not the specified abstract pathname size is
73       * strictly greater than the given number of k-bytes
74       * @param args size in KB
75       * @param filterLine line number
76       * @return true if and only if pathname size is strictly greater
77       *             than the given number of k-bytes
78       * @throws TypeIError WARNINGS
79       */
80      private static FileFilter greaterThan(String[] args, int filterLine) throws TypeIError {
81          try {
82              if (args.length != 1)
83                  throw new TypeIError(filterLine);
84              double size = Double.parseDouble(args[0]);
85              if (size < 0)
86                  throw new TypeIError(filterLine);
87              return pathname -> pathname.length()/BYTES_IN_KB > size;
88          } catch (Exception e) {
89              throw new TypeIError(filterLine);
90          }
91      }
92
93      /**
94       * Tests whether or not the specified abstract pathname size is
95       * between (inclusive) the given numbers (in k-bytes)
96       * @param args size bounds (inclusive)
97       * @param filterLine line number
98       * @return true if and only if pathname size is between (inclusive)
99       *             the given numbers (in k-bytes)
100      * @throws TypeIError WARNINGS
101      */
102     private static FileFilter between(String[] args, int filterLine) throws TypeIError {
103         try {
104             if (args.length != 2)
105                 throw new TypeIError(filterLine);
106             double lower = Double.parseDouble(args[0]);
107             double upper = Double.parseDouble(args[1]);
108             if (lower < 0 || upper < 0 || upper < lower)
109                 throw new TypeIError(filterLine);
110             return pathname -> pathname.length()/BYTES_IN_KB >= lower &&
111                                pathname.length()/BYTES_IN_KB <= upper;
112         } catch (Exception e) {
113             throw new TypeIError(filterLine);
114         }
115     }
116
117     /**
118      * Tests whether or not the specified abstract pathname size is
119      * strictly smaller than the given number of k-bytes
120      * @param args size in KB
121      * @param filterLine line number
122      * @return true if and only if pathname size is strictly smaller
123      *             than the given number of k-bytes
124      * @throws TypeIError WARNINGS
125      */
126     private static FileFilter smallerThan(String[] args, int filterLine) throws TypeIError {
127         try {
```

```
128             if (args.length != 1)
129                 throw new TypeIError(filterLine);
130             double size = Double.parseDouble(args[0]);
131             if (size < 0)
132                 throw new TypeIError(filterLine);
133             return pathname -> pathname.length()/BYTES_IN_KB < size;
134         } catch (Exception e) {
135             throw new TypeIError(filterLine);
136         }
137     }
138
139     /**
140      * Tests whether or not the specified abstract pathname last name
141      * equals the file name
142      * @param args file name to compare
143      * @param filterLine line number
144      * @return true if and only if the last name in the pathname
145      *            equals the given file name
146      * @throws TypeIError WARNINGS
147      */
148     private static FileFilter file(String[] args, int filterLine) throws TypeIError {
149         try {
150             String fileName = (args.length == 0) ? "" : args[0];
151             return pathname -> pathname.getName().equals(fileName);
152         } catch (Exception e) {
153             throw new TypeIError(filterLine);
154         }
155     }
156
157     /**
158      * Tests whether or not the given substring is contained in
159      * the last name of the specified abstract pathname
160      * @param args sub-string to search for
161      * @param filterLine line number
162      * @return true if and only if value is contained in the
163      *            file name (excluding path)
164      * @throws TypeIError WARNINGS
165      */
166     private static FileFilter contains(String[] args, int filterLine) throws TypeIError {
167         try {
168             String value = (args.length == 0) ? "" : args[0];
169             return pathname -> pathname.getName().contains(value);
170         } catch (Exception e) {
171             throw new TypeIError(filterLine);
172         }
173     }
174
175     /**
176      * Tests whether or not the given substring is the prefix
177      * of the file name (excluding path)
178      * @param args substring to check
179      * @param filterLine line number
180      * @return true if and only if value is the prefix of the
181      *            file name (excluding path)
182      * @throws TypeIError WARNINGS
183      */
184     private static FileFilter prefix(String[] args, int filterLine) throws TypeIError {
185         try {
186             String prefix = (args.length == 0) ? "" : args[0];
187             return pathname -> pathname.getName().startsWith(prefix);
188         } catch (Exception e) {
189             throw new TypeIError(filterLine);
190         }
191     }
192
193     /**
194      * Tests whether or not the given substring is the suffix
195      * of the file name (excluding path)
```

```
196         * @param args substring to check
197         * @param filterLine line number
198         * @return true if and only if value is the suffix of the
199         *            file name (excluding path)
200         * @throws TypeIError WARNINGS
201         */
202        private static FileFilter suffix(String[] args, int filterLine) throws TypeIError {
203            try {
204                String suffix = (args.length == 0) ? "" : args[0];
205                return pathname -> pathname.getName().endsWith(suffix);
206            } catch (Exception e) {
207                throw new TypeIError(filterLine);
208            }
209        }
210
211        /**
212         * Does file have writing permission? (for the current user)
213         * @param args permission
214         * @param filterLine line number
215         * @return true if and only if the file have writing permission
216         * @throws TypeIError WARNINGS
217         */
218        private static FileFilter writable(String[] args, int filterLine) throws TypeIError {
219            try {
220                String value = args[0];
221                if (!(value.equals("YES") || value.equals("NO")))
222                    throw new TypeIError(filterLine);
223                boolean permission = value.equals("YES");
224                return pathname -> (permission == pathname.canWrite());
225            } catch (Exception e) {
226                throw new TypeIError(filterLine);
227            }
228        }
229
230        /**
231         * Does file have execution permission? (for the current user)
232         * @param args permission
233         * @param filterLine line number
234         * @return true if and only if the file have execution permission
235         * @throws TypeIError WARNINGS
236         */
237        private static FileFilter executable(String[] args, int filterLine) throws TypeIError {
238            try {
239                String value = args[0];
240                if (!(value.equals("YES") || value.equals("NO")))
241                    throw new TypeIError(filterLine);
242                boolean permission = value.equals("YES");
243                return pathname -> (permission == pathname.canExecute());
244            } catch (Exception e) {
245                throw new TypeIError(filterLine);
246            }
247        }
248
249        /**
250         * Is the file a hidden file?
251         * @param args YES\NO
252         * @param filterLine line number
253         * @return true if and only if the file is hidden file.
254         * @throws TypeIError WARNINGS
255         */
256        private static FileFilter hidden(String[] args, int filterLine) throws TypeIError {
257            try {
258                String value = args[0];
259                if (!(value.equals("YES") || value.equals("NO")))
260                    throw new TypeIError(filterLine);
261                boolean permission = value.equals("YES");
262                return pathname -> (permission == pathname.isHidden());
263            } catch (Exception e) {
```

```
264                 throw new TypeIError(filterLine);
265             }
266         }
267 
268         /**
269          * all files are matched
270          * @return true for every pathname
271          */
272         private static FileFilter all(String[] args, int filterLine) throws TypeIError {
273             if (args.length != 0)
274                 throw new TypeIError(filterLine);
275             return pathname -> true;
276         }
277     }
278     // End of FilterFactory
```

8 filesprocessing/Filter/NegatedFilter.java

```java
package filesprocessing.Filter;

import java.io.File;
import java.io.FileFilter;

/**
 * Decorator class for FileFilter
 */
public class NegatedFilter implements FileFilter {

    // ###################### //
    // #### ATTRIBUTES #### //
    // ###################### //

    /** The original filter to be negated */
    private final FileFilter originalFilter;

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    /**
     * constructs a new negated filter
     * @param filter filter to be negated
     */
    NegatedFilter(FileFilter filter) {
        originalFilter = filter;
    }

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * Tests whether or not the specified abstract pathname should
     * be included in a pathname list.
     * @param pathname The abstract pathname to be tested
     * @return true if and only if pathname    should be included
     */
    @Override
    public boolean accept(File pathname) {
        return !originalFilter.accept(pathname);
    }
}
// End of NegatedFilter Decorator class
```

9 filesprocessing/Order/OrderFactory.java

```java
package filesprocessing.Order;

import filesprocessing.ProcessingExceptions.TypeIError;

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

/**
 * A factory class for orders used in Directory Processor
 */
public class OrderFactory {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    private static final int EQUAL = 0;
    private static final String PERIOD_DELIMITER = ".";

    private static final String ABSOLUTE_NAME_ORDER = "abs";
    private static final String TYPE_ORDER = "type";
    private static final String SIZE_ORDER = "size";

    // ##################### //
    // #### ATTRIBUTES #### //
    // ##################### //

    // ###################### //
    // #### CONSTRUCTOR #### //
    // ###################### //

    // ################## //
    // #### METHODS #### //
    // ################## //

    /**
     * Returns the selected file order
     * @param orderName order name
     * @param orderLine line number
     * @param reversedOrder is the order reversed?
     * @return The selected file order
     * @throws TypeIError WARNINGS
     */
    public static Comparator<File> select(String orderName, int orderLine,
                                          boolean reversedOrder) throws TypeIError {
        Comparator<File> order;
        try {
            switch (orderName) {
            case ABSOLUTE_NAME_ORDER:
                order = abs();
                break;
            case TYPE_ORDER:
                order = type();
                break;
            case SIZE_ORDER:
                order = size();
                break;
            default: // order name not matching any case
                throw new TypeIError(orderLine);
```

```
60              }
61              // check if order is reversed
62              return (reversedOrder) ? order.reversed() : order;
63          } catch (TypeIError e) {
64              throw new TypeIError(orderLine);
65          }
66      }
67
68      // #### ORDERS OF THE PROCESSOR #### //
69
70      /**
71       * File order according to its absolute name, from 'a' to 'z'.
72       * @return a file comparator according to absolute name.
73       */
74      private static Comparator<File> abs() {
75          return Comparator.comparing(File::getAbsolutePath);
76      }
77
78      /**
79       * File order by type, then by absolute name.
80       * @return a file comparator by type, then by absolute name.
81       */
82      private static Comparator<File> type() {
83          return (file1, file2) -> {
84              int order = fileType(file1).compareTo(fileType(file2));
85              if (order == EQUAL)
86                  return abs().compare(file1, file2);
87              return order;
88          };
89      }
90
91      /**
92       * File order by size, then by absolute name.
93       * @return a file comparator by size, then by absolute name.
94       */
95      private static Comparator<File> size() {
96          return (file1, file2) -> {
97              int order = Long.compare(file1.length(), file2.length());
98              if (order == EQUAL)
99                  return abs().compare(file1, file2);
100             return order;
101         };
102     }
103
104     /**
105      * Returns the file type (i.e. its extension)
106      * @param file the file to check
107      * @return the file extension.
108      */
109     private static String fileType(File file) {
110         int delimiterIDX = file.getName().lastIndexOf(PERIOD_DELIMITER);
111         /* I. (delimiterIDX == -1) NOT_FOUND
112          * A file without a period in its name is considered to have
113          * the empty string as its extension.
114          * II. (delimiterIDX == 0)
115          * A file starts with a period and its name does not contain another
116          * period (as delimiter), it should be treated as if the file's
117          * type is the empty string. */
118         if (delimiterIDX < 1)
119             return "";
120         int endIDX = file.getName().length();
121         return file.getName().substring(delimiterIDX, endIDX);
122     }
123 }
124 // End of OrderFactory
```

10 filesprocessing/ProcessingExceptions/ProcessingExc

```java
package filesprocessing.ProcessingExceptions;

/**
 * Super class for Processing Exceptions
 */
public class ProcessingExceptions extends Exception {

    /**
     * A constructor delegates its actions to the Exception class
     * constructor.
     * @param errorMessage The error message.
     */
    public ProcessingExceptions(String errorMessage) {
        super(errorMessage);
    }
}
```

11 filesprocessing/ProcessingExceptions/TypeIError.jav

```java
package filesprocessing.ProcessingExceptions;

/**
 * Type I errors (Warnings):
 * Bad parameters in the FILTER/ORDER line
 */
public class TypeIError extends ProcessingExceptions {

    /**
     * Exception Constructor
     * @param lineNum number of line where the warning had be thrown
     */
    public TypeIError(int lineNum) {
        super(String.format("Warning in line %d", lineNum));
    }
}
```

12 filesprocessing/ProcessingExceptions/TypeIIError.ja

```java
package filesprocessing.ProcessingExceptions;

/**
 * Type II errors:
 * Invalid Usage, I/O problems, bad sub-section name
 */
public class TypeIIError extends ProcessingExceptions{

    /**
     * Exception Constructor
     * @param errorMessage Error message
     */
    public TypeIIError(String errorMessage) {
        super("ERROR: " + errorMessage);
    }
}
```