Object Oriented Programming - Exercise 5: Files Processing

Contents

1 Goals **3**

2 Overview **3**

3 Package **3**

4 Commands File Structure **4**
- 4.1 FILTER . . . 4
  - 4.1.1 #NOT suffix . . . 5
  - 4.1.2 Notes . . . 5
- 4.2 ORDER . . . 6
  - 4.2.1 #REVERSE suffix . . . 6
  - 4.2.2 Sorting . . . 6
  - 4.2.3 Comments . . . 7

5 Output **7**

6 Error Handling **8**
- 6.1 Introducing The Error Types . . . 8
  - 6.1.1 Type I Errors - Warnings . . . 8
  - 6.1.2 Type II Errors . . . 9
- 6.2 Examples . . . 9
- 6.3 General Remarks . . . 9
- 6.4 Simplifying Assumptions . . . 10

7 Design **10**

8 README **10**

9 Misc. **11**
- 9.1 School Solution . . . 11
- 9.2 Testing . . . 11
- 9.3 External packages . . . 11

10 Submission Details **11**
10.1 Deadline 11
10.2 Jar File 11

1 Goals

- Apply design principles and design patterns that were learned in the lectures
- Working with file attributes
- Working with directories and directory structures
- Working with the Exceptions mechanism

2 Overview

In this exercise you will implement a flexible framework for working with files. Your program will **filter** the files in a given directory according to various conditions, and **order** the filenames that passed the filtering according to various properties. You will implement a program called DirectoryProcessor (which will be part of a package called filesprocessing), which is invoked from the command line as follows:

java filesprocessing.DirectoryProcessor sourcedir commandfile

Where:

1. **sourcedir** is a directory name, in the form of a path (e.g., "./myhomeworks/homework1/" or "./myhomeworks/homework1"). This directory is referred to in the following as **Source Directory**. **sourcedir** can be either absolute or relative to where we run the program from.
2. **commandfile** is a name of a file, also in the form of a relative or absolute path (e.g., ./scripts/Commands1.txt). This file is referred to in the following as **Commands File**. It is a text file that contains sections, wherein each section contains a FILTER and an ORDER subsections (see section 4). The FILTER sub-section includes filters which are used to select a subset of the files **in the Source Directory** .The ORDER sub-section indicates in which order the files' names should be printed.

3 Package

As you saw above, *DirectoryProcessor* should be placed under the *filesprocessing* package. By default any other class you implement in this exercise should be in the *filesprocessing* package unless you decide to add more packages/sub-packages of your own (which we also encourage you to do).

4 Commands File Structure

This text file is composed of one or more sections. **Each** section is composed of the following two sub-sections:

1. FILTER
2. ORDER

Both sub-sections **must** appear in every section.

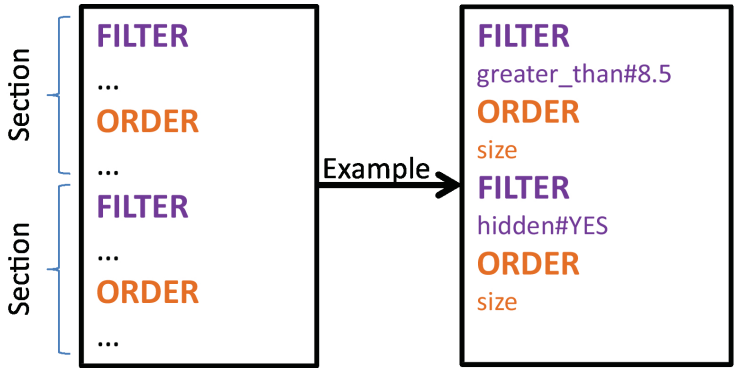


Figure 1: The **Commands File** is composed of sections. Each section has two sub-sections.

Note - multiple FILTER+ORDER sections may appear in the same file. Different sections should appear one after the other, without any empty lines separating them. Each section should be handled separately.

4.1 FILTER

This part describes the filters that will be used in the program. Filters will return all files in **Source Directory** that match a certain criterion. Only files are returned (not directories). Only files that are directly under the source directory are returned (files that are in directories that are under the source directory should **not** be returned). Each filter sub-section starts with a line which consists of the word **FILTER**, followed by **a single** line describing the filter. A filter is a condition; it is satisfied by some files (possibly none).

We start by describing the filters. The format for each filter is either NAME, NAME#VALUE or NAME#VALUE#VALUE. For simplicity, you may assume that both NAME and VALUE are strings that contain only letters (uppercase or lowercase), digits, and the following characters: "/", ".", "-", "_" without any spaces or other symbols, **except** for the following cases - the VALUE given in the "file","contains","prefix","suffix" filters (and only in them) **can** contain spaces.

Filter Name	Meaning	Value format	Example
greater_than	File size is strictly greater than the *given* number of k-bytes	double	greater_than#5
between	File size is between (inclusive) the *given* numbers (in k-bytes)	double#double	between#5#10
smaller_than	File size is strictly less than the *given* number of k-bytes	double	smaller_than#50.5
file	*value* equals the file name (excluding path)	string	file#file.txt
contains	*value* is contained in the file name (excluding path)	string	contains#ile
prefix	*value* is the prefix of the file name (excluding path)	string	prefix#aaa
suffix	*value* is the suffix of the file name (excluding path)	string	suffix#.txt
writable	Does file have *writing* permission? (for the current user)	YES or NO	writable#YES
executable	Does file have *execution* permission? (for the current user)	YES or NO	executable#NO
hidden	Is the file a hidden file?	YES or NO	hidden#NO
all	all files are matched	-	all

4.1.1 #NOT suffix

1. Each filter may appear with the trailing #NOT suffix. This means that this filter satisfies exactly all files not satisfied by the original filter. For example, *greater_than*#100#NOT satisfies all files that are not greater than 100 k-bytes (i.e., files that are smaller than or equal to 100 k-bytes).
2. The #NOT suffix may only appear once per filter. You are not required to support more complex cases (e.g., inputs such as prefix#a#NOT#NOT), and may support them or consider them as error, as you prefer (your program will not be tested on such cases). You may also assume that the #NOT suffix always comes after some filter (i.e., you will not be tested on filter lines such as "#NOT").

4.1.2 Notes

1. As written above, a FILTER sub-section must appear in every section. Otherwise it is an error - see section 6.1.2.
2. You may assume that there is **only** one filter after the FILTER line.
3. Sub-directories may appear in the **sourcedir** directory - you should ignore them and treat only files.
4. You may assume filter format is always NAME#VALUE or NAME#VALUE#VALUE (in the case of *between* filter) or just NAME (in the case of *all* filter).
5. The domain for size filters (*smaller_than, between* and *greater_than*) is any non-negative double number (java double, greater than or equal to 0), which may or may not contain a fractional part (i.e., either 5, 5., 0, 124, etc., or 0.111, 532.5, .5, etc.). You may assume that the input is a java *double* but should **verify** it is a non-negative *double*.
6. *between* filter receives 2 values separated by #. For example, between#10#20 should return all files with size greater than (or equal to) 10 and smaller than (or equal to) 20. You should **validate** that the first value is smaller or equal to the second. (i.e., input such as between#13#10 is considered illegal, see section 6.1.1).
7. Conversion between bytes and k-bytes is straightforward: 1 kb = 1024 bytes.
8. For the *writable/executable/hidden* filters, the domain is YES/NO strings. You **need** to verify this. Other values are considered errors, see section 6.1.1.
9. For *file, prefix, suffix* and *contains* filters, the domain is any string composed of the legal characters described above. You may assume that the input values for these filters do not

contain illegal characters.

(a) *file* filter matches file names equal to the filter value. For example, *file*#a.txt matches files called "a.txt". Comparison is **case-sensitive** (i.e., "a.txt" is not matched by "A.txT").

(b) *contains* filter matches file names that contain the filter value. For example, *contains*#abc matches files that have "abc" in their name. Comparison is case-sensitive (i.e., "oop_abcd.txt" is **not matched** by "oop_ABCd.txt.

(c) *prefix/suffix* filters match file names that start/end with the filter value respectively. For example, the *prefix* "aa" matches any file name that starts with "aa" (e.g., aa, aa1, aa123a.txt, etc.). Similarly, the *suffix* ".txt" matches any file name that ends with ".txt" (e.g., .txt, a.txt, f2b.txt, etc.). These filters are **case-sensitive** as well.

(d) All textual comparisons are excluding the path. "Excluding path" means that the string can be found in the file name itself and not in other parts of the string that describes its path. For example, *contains*#*abc* will match the file "/cs/files/abc2.txt" but will not match the file "/cs/abc/files/hello.txt".

4.2 ORDER

This sub-section indicates the order in which the filtered files are printed. The following are possible orders:

Order Name	Meaning
abs	Sort files by absolute name (using File.getAbsolutePath()), going from 'a' to 'z'
type	Sort files by file type, going from 'a' to 'z'
size	Sort files by file size, going from smallest to largest

4.2.1 #REVERSE suffix

1. Each order may appear with the trailing #REVERSE suffix. This means that the files should be printed in the opposite way of the original order. For example, *size*#REVERSE should print the files from largest to smallest.
2. The #REVERSE suffix may only appear once per order. You are not required to support more complex cases (e.g., inputs such as abs#REVERSE#REVERSE), and may support them or consider them as error, as you prefer (your program will not be tested on such cases). You may also assume that the #REVERSE suffix always comes after some order (i.e. you will not be tested on filter lines such as "#REVERSE").

4.2.2 Sorting

In order to print the file names in the designated order, they must be sorted first. You must sort the files using a sorting method that **you implement yourself**. You may choose to implement any algorithm you want. Full credit will only be given to **efficient** sorting algorithms. You may not use any of Java's built-in sorting methods.

4.2.3 Comments

1. Each order sub-section starts with a line which consists of the word **ORDER**, followed by **at most** a single line describing the order.
2. As written above, an ORDER sub-section must appear in every section. Otherwise it is an error.
3. An ORDER sub-section may be empty (i.e. the line containing the word "ORDER" is the last line in the file or is followed by a new "FILTER" line of a new section). In case the sub-section is empty, the *abs* order should be used.
4. For string orders (*abs* and *type*), use the String.compareTo() method to compare two file names.
5. In case two or more files are equal according to any of the *type* and *size* orders, the *abs* order should be used to order them. For example, in *size* order, two files with the same size should be ordered by their absolute name.
6. A file without a period in its name is considered to have the empty string as its extension, i.e. when ordering by type, "file" is considered equal to "file." (this is not the case when ordering by abs).
7. In case there is more than one period in the file name, you should treat the last one as the delimiter between the name and the type. (e.g. file.1.txt is of type txt). Please note that if a file starts with a period and its name does not contain another period (as delimiter), it should be treated as if the file's type is the empty string (a period at the start of a filename represents a hidden file).

5 Output

The names of the filtered files in a current section should be **printed** line by line in the order specified in the **ORDER** sub-section. The printed names should only include the file names, excluding the whole path. Consider the following example: two files named *file.txt* and *file1.txt* exist in the folder **Source Directory**. *file.txt* has no write permission and its size is 12 bytes, while *file1.txt* is writable and its size is 6 bytes. You can see below the **Commands File** and the desired output.

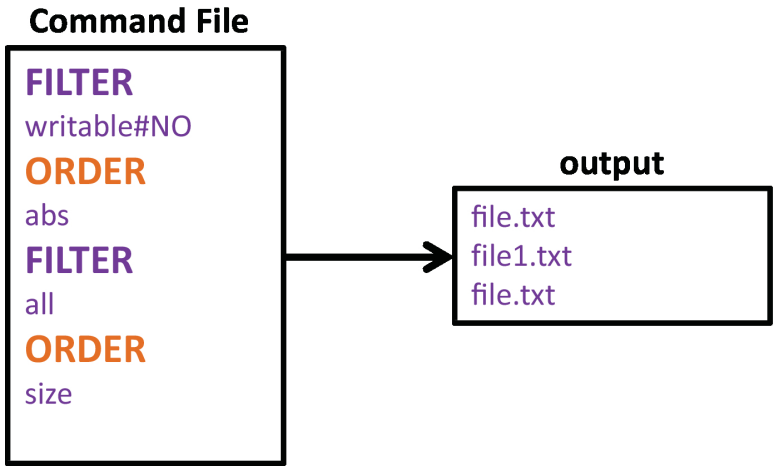


Figure 2: The first *file.txt* in the output is the result of the first section. The two next lines are the result of the second section.

6 Error Handling

You are required to use the exceptions mechanism to handle errors in your program. Correctly defining and using the different exception classes is a major part of this exercise. You should divide the potential errors that might occur in your program to hierarchical groups. Below we summarize the two types of potential errors. In the first section (type I errors), you catch the error, print a warning message and continue normally. In the second section (type II errors), you are required to catch the error and return.

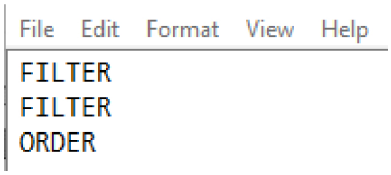
6.1 Introducing The Error Types

6.1.1 Type I Errors - Warnings

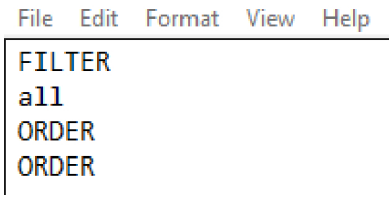
1. A bad FILTER/ORDER name (e.g., *greaaaater_than*). These names are also case-sensitive (e.g., *Size* is an **illegal** order name, and should result in an error).
2. Bad parameters to the *hidden/writable/executable* filters (anything other than YES/NO).
3. Bad parameters to the *greater_than/between/smaller_than* filters (negative number).
4. Illegal values for the between filter (for example - *between*#15#7).

Comments

- Type I errors should result in printing "Warning in line X" (print to the standard error using System.err) and continuing normally. X is the line number where the FILTER/ORDER problem occurred, where 1 is the line number of the first line. All warnings are printed together, before printing the matched file names (more details in section 6.3).
- In case there is a warning in the **FILTER** sub-section, you should behave as if the filter was *all* (i.e.,you should match all files).
- In case a type I error occurs in the **ORDER** sub-section, you should behave as if there was no order specified (i.e., order by *abs*).
- In particular, you should handle this **Commands File**:



  as follows: First print "Warning in line 2". Afterwards print all the files in the **sourcedir** with the abs order (the second FILTER should be interpreted as a bad FILTER name).
- Similarly, you should handle this **Commands File**:



  as follows: First print "Warning in line 4". Afterwards print all the files in the **sourcedir** with the abs order (the second ORDER should be interpreted as a bad ORDER name).

- If two warnings exist in the same line (i.e., *between#2#-1*), you should print only one warning, as you choose.

6.1.2 Type II Errors

5. Invalid usage (i.e., anything other than two program arguments, where the first is the **Source Directory** and the second is the **Commands File**). You may assume **Source Directory** is an existing directory and **Commands File** is an existing file.
6. I/O problems - errors occurring while accessing the **Commands File**.
7. A bad sub-section name (i.e., not FILTER/ORDER). Sub-section names are **case-sensitive** (e.g., filter is an illegal sub-section name, and should result in an error).
8. Bad format of the Commands File (e.g., no ORDER sub-section).

Comments

- Type II errors should result in printing "ERROR: " to stderr (using System.err), afterwards an informative message (with newline) and then returning (Error format - "ERROR: <error message> \n"). Please refrain from exclamation marks in these error messages. You are allowed to catch type II errors in the main method and handle them there.
- Upon any error in the **Commands File** (Type II errors), your program should **not print any file names** or any warnings (Type I errors). This includes a file with two sections, where the first section is valid and the second isn't.

6.2 Examples

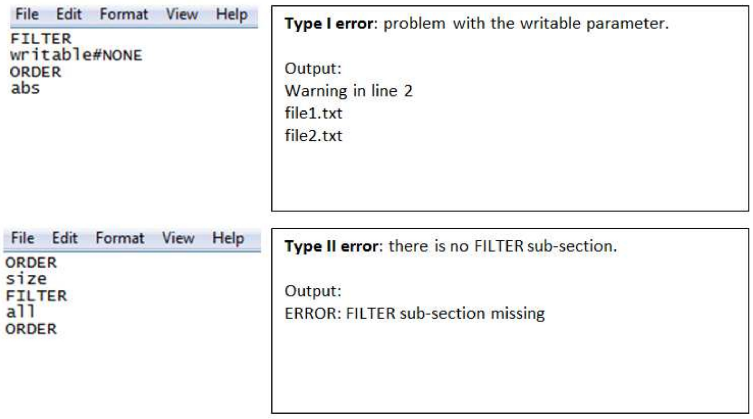


Figure 3: Example for the two types of errors. In the first example the program prints a warning caused by an illegal parameter ("*writable*"). It then prints the matched files (all files in this case, since the filter is illegal) in the given ORDER. In the second example the program prints ERROR with an informative message, without printing any matched files because a sub-section FILTER is missing in the first section.

6.3 General Remarks

1. Warnings should be printed in the same order they appear in.

2. Printing the matched files should be done **after** printing warnings (type I errors) from all sub-sections. Each section's warnings should be printed before printing its matched files. That is, the order should be:

   Print warnings of section 1
   Print matched files of section 1
   Print warnings of section 2
   Print matched files of section 2
   ...

6.4 Simplifying Assumptions

Operating systems and file systems can be quite complex, and real file management programs need to deal with this complexity. To simplify the task, you can make the following assumptions, and your solution does not need to check that they hold:

1. The file system (under the **Source Directory**) is a real tree (no hard or symbolic links).
2. File names do not include special symbols, only letters (A-Z, a-z), digits (0-9), ".", "_", "-" and white spaces. However, filtered files may be located in directories that do contain other characters. For example, in source dir /aaa#3/, the file /aaa#3/work.txt/ may be filtered.
3. No other process changes the files under the **Source Directory** while your program is running.
4. Regarding white spaces in the input: you may assume there are no redundant white spaces anywhere in the **Commands File**. This includes trailing or preceding white spaces, line-break, etc. You may ignore such cases in your program, or handle them as you see fit.

7 Design

Your program should follow the design principles you have learned so far: modularity, the factory design pattern and the single choice principle. **Explain your chosen design in detail in your README!**

8 README

Please address the following points in your README file:

1. Explain all your design choices in detail.
2. Describe the exceptions hierarchy you used in order to handle errors in the program. Explain the considerations that made you choose that specific design.
3. How did you sort your matched files? Did you use a data structure for this purpose? If so, what data structure and why?

9 Misc.

9.1 School Solution

The school solution can be found in ∼oop/bin/ex5/ex5SchoolSolution. You are highly encouraged to use the school solution to check if/how you need to handle each case or parameter. Your output should be **exactly** the same as the school solution, aside from error messages following "ERROR: " in case of type II errors, though you still have to make sure these contain no line-breaks. However, if you see some clearly unintended behavior in the school solution, please verify it with the course staff. You are encouraged to use the published automatic tests to experiment with the school solution before starting to work on your code, in order to get a feeling of how your program should behave.

9.2 Testing

As with previous exercises, you are encouraged to test your own code using all the tools you've learned. You should not submit the tests you write. They are for your personal use.

9.3 External packages

You may use the classes available under packages java.util.*, java.util.function, java.lang.*, java.io.* and java.text.* in standard java 1.8 distribution. Specifically, in this exercise you are **allowed to change the java API level to 8**. You are advised to use the **java.io.File** class for reading directories, and **String** for working with text. You are highly encouraged to examine the full API of each class you use. If you wish, you may also use classes from the *java.nio* package (although it is your responsibility to carefully read the API of this package and use it correctly). Apart from that you **may not** use any other class that you didn't write yourself.
Also remember you are not allowed to use any built-in sorting function of Java. You must implement your own sorting function.

10 Submission Details

10.1 Deadline

The exercise is due by **December 30, 2020**

10.2 Jar File

You should submit a file named ex5.jar containing all the *.java* files of your program, as well as the README. Please note the following:

- Files should be submitted in the directory hierarchy of their original packages.
- No *.class* files should be submitted.
- There is no need to submit any testers.
- Your program must compile without any errors or warnings.
- Javadoc should compile correctly.

To compile your code in command line with the correct warning testing:

javac -Xlint:rawtypes -Xlint:static -Xlint:empty -Xlint:divzero -Xlint:deprecation

In order to have them displayed when you compile your code (in IDEA), go to `File->Settings->Build, Execution, Deployment->` *Compiler*`->Java Compiler` and enter the following line under `Additional command line parameters`:

-Xlint:rawtypes -Xlint:static -Xlint:empty -Xlint:divzero -Xlint:deprecation

If you decided to create an additional package (e.g. "orders") use the following command to create the jar file:

jar -cvf ex5.jar README filesprocessing/.java orders/*.java*

This command should be run from the main project directory (the one that contains the *filesprocessing* and *orders* directories).

Good-Luck!