Exercise 5 – File Processing

OOP 2015

1 Goals

1. Working according to a given design
2. Working with file attributes
3. Working with directories and directory structures
4. Working with the Exceptions mechanism

2 Submission Details

- Submission Deadline: **Sunday, 07/06/2015, 23:55**
- You may use the classes available under packages java.util.*, java.lang.*, java.io.* and java.text.* in standard java 1.7 distribution. 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.

3 Introduction

In this exercise you will implement a flexible framework for working with files. Your program will be able to **filter** certain files from a given directory, and **print** the names of these files in a certain **order**. Specifically, in this exercise you will implement a program, called MyFileScript (which will be part of a package called filescript), and is invoked from the command line as follows:

java filescript.MyFileScript sourcedir commandfile

Where:

1. **sourcedir** is a directory name, in the form of a path (e.g., "./myhomeworks/homework1/"). This directory is referred to in the following as the **Source Directory**. **sourcedir** can be either absolute (starting with "/" in Linux or a drive letter in Windows) or relative to where we run the program from.

2. **commandfile** is a name of a file, also in the form of a path (relative or absolute. E.g., ./scripts/Commands1.txt). This file is referred to in the following as the **Commands File**. It is a text file that contains couples of FILTER/ORDER sub-sections (see figure 1). The FILTER sub-section includes filters which are used to select a subset of the files. The ORDER sub-section indicates in which order the files' names should be printed.

3.1 Package

As you see above, *MyFileScript* should be placed under the *filescript* package. By default any other class you implement in this exercise should be in the *filescript* package unless you decide to add more 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 (see figure 1). **Each** section is composed of the following two sub-sections:

1. FILTER
2. ORDER

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

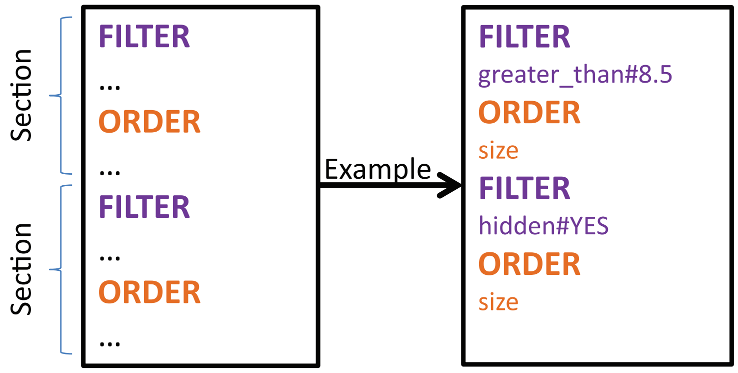


Figure 1: The **Commands file** is composed of sections. Each section has two sub-sections.

4.1 FILTER

This part describes the filters that will be used in the program. Filters will search for files in the **Source Directory**, and return all files that match. 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 both NAME and VALUE are strings that may contain only letters (uppercase or lowercase), digits and the following characters: “/”, “.”, “-”, “_” without any spaces or other symbols.

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	*string* equals the file name (excluding path)	string	file#file.txt
contains	*string* is contained in the file name (excluding path)	string	file#ile
prefix	*string* is the prefix of the file name (excluding path)	string	prefix#aaa
suffix	*string* 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 file a hidden file?	YES or NO	hidden#NO
all	all files are matched	-	all

4.2 #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").

4.3 Comments

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 the *all* filter.
5. The domain for size filters (*smaller_than, between* and *greater_than*) is any non-negative double number (java double, equal or greater than 0), which may or may not contain a fractional part (i.e., either such as 5, 0, 124, etc., or such as 0.111,

532.5, 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 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 (e.g., “a.txt” is not matched by the prefix “A”). The *suffix* ‘at’ will capture both ‘hello.sat’ and ‘bye.cat’ but will not capture ‘matan.txt’.
   (d) “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’.

4.4 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 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

4.5 #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, at 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").

4.6 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 - see section 6.1.2.
3. An ORDER sub-section may be empty (i.e., the line with the word **ORDER** and no other lines afterwards). 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. In the *abs* order, capital letters will be ordered **before** the lowercase letters. (e.g., A.txt should be printed before a.txt).
7. You can assume that there will always be a suffix indicating a file type (e.g. file1.txt or file1.mat etc...)
8. 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).

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 **Command File** and the desired output.

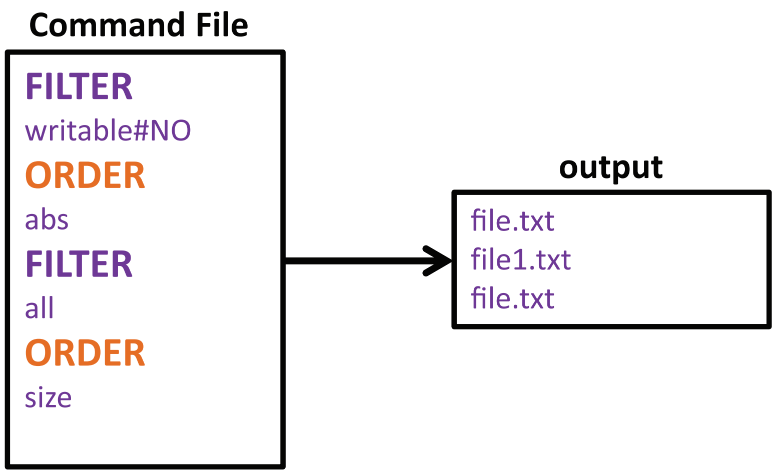


Figure 2: The first *file.txt* in the output is the result of the first section. The two next lines are the result of the second section.

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 exit.

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 (*between*#15#7).

Comments

- Type I errors should result in printing "Warning in line X" (standard error - System.err) and continuing normally (X is the line number where the FILTER problem occurred, where 1 is the line number of the first line). All warnings are printed together, before printing the matched file names.
- In case there is a warning in the **FILTER** sub-section which causes the sub-section to be empty, 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*).
- If two warnings exist in the same line (i.e., *between*#2#-1), you should print only one warning .

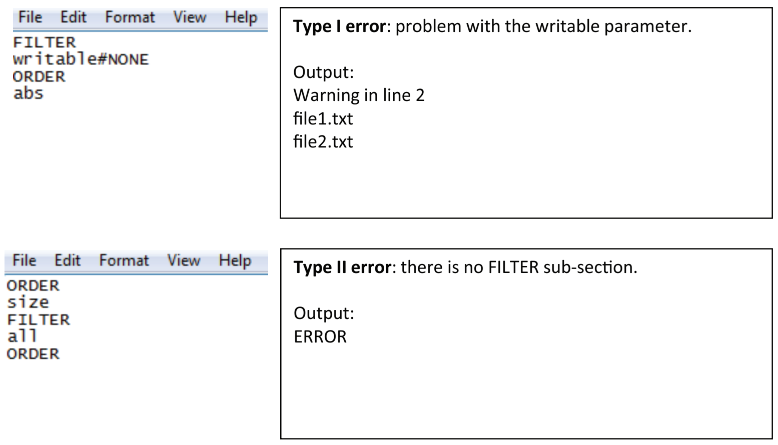
6.1.2 Type II Errors

5. Invalid usage (i.e., anything other than 2 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" (with newline) to STDERR (using System.err) and exiting the program. You are allowed to catch type II errors in the main file and handle them there.
- Upon any error in the **Commands File** (Type II errors), your program should **not print any file name** or any warning (type I errors). This includes a file with 2 sections, where the first section is proper and the second isn't.

6.2 Examples



```
FILTER
writable#NONE
ORDER
abs
```

Type I error: problem with the writable parameter.

```
Output:
Warning in line 2
file1.txt
file2.txt
```

```
ORDER
size
FILTER
all
ORDER
```

Type II error: there is no FILTER sub-section.

```
Output:
ERROR
```

Figure 3: Example for the two types of errors. In the first example the program prints warning caused by an illegal *writable* parameter. Then it prints the matched files (all files in this case, since the filter is illegal). In the second example the program simply prints ERROR without printing matched files because a sub-section FILTER is missing in the first section.

6.3 General Remarks

1. Recall that multiple FILTER+ORDER sections may appear in the same file. Different sections should appear one after the other, without any line separating them. Each section should be handled separately.
2. Warnings should be printed in the same order they appear in.
3. Printing the matched files should be done **after** printing warnings (type I errors) from all sub-sections. In case of multiple sections in the **Commands file** each section's warnings should be printed before performing its matched file printing. 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 spaces or special symbols, only letters (A-Z, a-z), digits (0-9), '.', '_' and '-'. 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, etc. You may ignore such cases in your program, or handle them as you see fit.

7 Design

Your program should follow the design shown in Tirgul 9. If you choose to work with a different design, you must explain it in detail in your README file. In your explanation, you should specify why you chose that design, why it is preferable to the design presented in class, and what are the downsides of using it. Implementing a different design without providing a proper explanation will result in a serious point reduction.

8 Submission Guidelines

8.1 README

Please address the following points in your README file:

1. Explain any non-trivial implementation details of your design (Either the one shown in class or your own)
2. Describe the Exception hierarchy you used in order to handle errors in the program. Explain the considerations that made you choose that specific implementation.
3. How did you sort your matched files? Did you use a data structure for this purpose? If you did, why choosing this particular data structure?

8.2 QUESTIONS

A part of this exercise it to answer some questions about the course material in weeks 8-9. The questions themselves are in the QUESTIONS link in Moodle. Write your answers directly in the file you downloaded and include it in your jar.

8.3 Jar File

You should submit a file named ex5.jar containing all the *.java* files of your program, as well as the README and QUESTIONS file. 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 with warning testing, use:

```
javac –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 files:

```
jar –cvf ex5.jar README QUESTIONS filescript/*.java orders/*.java
```

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

9 Misc.

9.1 School Solution

A school solution can be found in ∼oop/bin/ex5SchoolSolution You are highly encouraged to use the school solution to check if/how you need to handle each case or parameter. Your output on all cases should be exactly the same as the school solution's. However, if you see some clearly unintended behavior in the school solution, please verify it with 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 Automatic Tests

As usual, 90% of the automatic grade for the exercise is based on visible tests which will run on your code upon submission. The tests include both the filter files, and source directories. You can download from here the test files to your computer and use them to test your code locally

There are two folders, each containing filter files:

1. basic_filters/ - simple tests of the filters and orders.
2. advanced_filters/ - more advanced testing of the filters and orders.

There are two source directories to work with:

1. basic_source_directory/
2. advanced_source_directory/

Our automatic tests run each of the filters in each filter directory with the corresponding source directory. For example, the filter **basic**_filters/filter001.flt runs with **basic**_source_directory/. When you submit your exercise, we will run all the automatic tests presented above. Error messages of the pre-submission script will contain the filter number. For example, if you get the following error:

```
runTests[27](Ex5Tester): problem in test number:028, line number:1 expected:<[file2.txt]>
                         but was:<[Warning in line 2]>
```

This means that your program generated an output different than the school solution's when tested with the filter basic_filters/filter028.flt on basic_source_directory/. In order to fix such bugs, you can manually run the school solution against the specified test and compare its output with yours.

You can also (and are encouraged to) run the automatic tests from the terminal by running the following command:

```
~oop/bin/ex5 ex5.jar
```

Good-Luck