Exercise 6 – File Processing

OOP 2014

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: **Thursday, 02/06/2014, 23:55**
- This exercise will be done alone (**not in pairs**).
- 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. Apart from that you **may not** use any other class that you didn't write yourself. 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).

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 is invoked from the command line as follows:

java oop.ex6.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 "/") or relative to where we run the program from (starting with a different character).
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 subsections (see figure 1). The FILTER subsection includes filters which are used to select a subset of the files. The ORDER subsection indicates in which order the files' names should be printed.

3.1 Package

As you see above, *MyFileScript* and any other class you implement in this exercise should be placed under the *oop.ex6.filescript* package. You are encourage to build some other sub-package under the *ex6* package.

4 Commands File structure

This 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

These sub-section **must** appear in every section.

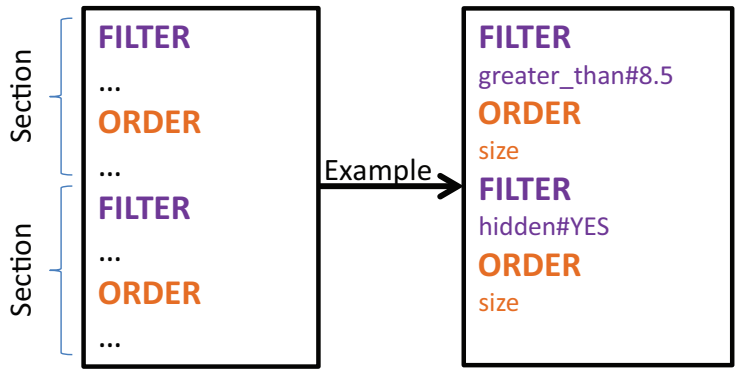

Figure 1: The **Commands file** is composed of sections. Each section has two sub-sections.

4.1 FILTER

This subsection describes the filters that will be used in this section. Filters will search for files in the **Source Directory**, and return all files that match. Only files are returned (not directories). Each filter subsection starts with a **FILTER** line 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. Each filter (except for one) is composed of a NAME#VALUE or NAME#VALUE#VALUE format. For simplicity, you may assume both NAME and VALUE are strings that may contain only letters (capital or minuscule), digits and the following characters: "/", ".", "-", "_" without any spaces or other symbols.

Filter Name	Meaning	Value format	Example
greater_than	File size is more than the *given* number of k-bytes	double	greater_than#5
between	File size is between (including) the *given* numbers (in k-bytes)	double#double	between#5#10
smaller_than	File size is 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, at 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 referred only to the files.
4. You may assume filter format (except the *all* filter) is always NAME#VALUE or NAME#VALUE#VALUE in the case of *between* filter (with no other suffix).
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., both inputs such as 5, 0, 124, etc., and inputs 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 10 but smaller than 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 as an error – 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* and *suffix* and *contains* filters, domain is any string, containing the characters described above.

   (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.). Search is also **case-sensitive** (e.g., “a.txt” is not matched by the prefix “A”).

4.4 ORDER

This subsection 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 be result printing 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. As written above, a ORDER sub-section must appear in every section. Otherwise it is an error - see section 6.1.2.
2. For string orders (*abs* and *type*), use the String.compareTo() method to compare 2 file names.
3. You may assume every order subsection contains at most a single (and not more) order line.
4. 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.
5. In the *abs* order, capital letters will be ordered **before** the lowercase letters. (e.g., A.txt should be printed before a.txt).
6. An ORDER sub-section may be empty (i.e., the ORDER line with no other lines). In case it does not appear, the *abs* order should be used.
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 dot 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 (the match all filters in **FILTER** section) in a current section should be **printed** line by line in the order from **ORDER** sub-section. The printed names is only the file name 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.

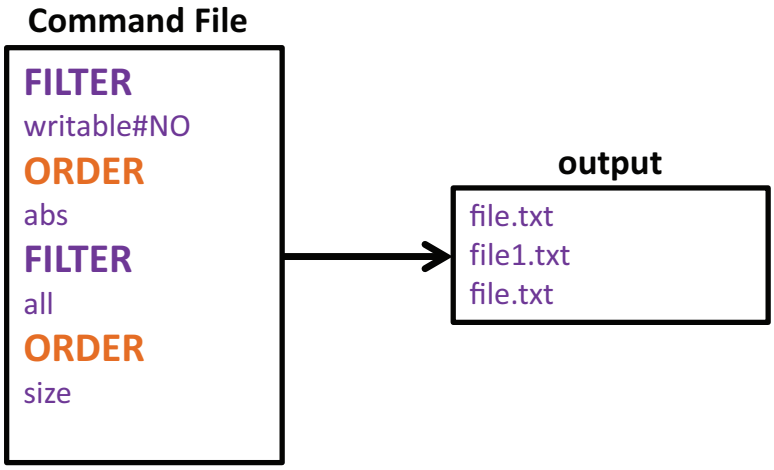

Figure 2: The first *file1.txt* in the output is referred to the result of the first section. The two next lines are the result of the second section.

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 could be invoked by your program to different error types organized in hierarchical groups, as shown in class. 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 (a zero or 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 output - System.out) and continuing normally, skipping these errors (X is the line number where the FILTER problem occurred). All warnings are printed together, before printing the matched files names.
- In case there is a warning in the **FILTER** sub-section which causes the sub-sections to be empty, you should behave as it like *all* filter (i.e.,you should match all files).
- In case Type I errors occur in the **ORDER** sub-section, you should behave as if there is no specify order (i.e., order by *abs*).
- If two warnings exist in the same line (i.e., *between*#2#-1), you should print the warning comment just once.

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 a valid **Commands File**). You may assume **Source Directory** is an existing directory and **Commands File** is an existing file.
6. I/O problems - error occurring while accessing to the **Command file**.
7. A bad subsection name (i.e., not FILTER/ORDER). Subsection names are **case-sensitive** (e.g., filter is an illegal subsection name, and should result in an error).
8. Bad order of Commands File. E.g., no ORDER subsection.

Comments

- Type II errors should result in printing "ERROR" (with newline) to STDERR (using System.err) and exiting the program. You are allowed to let the main file catch all Type II errors exceptions 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

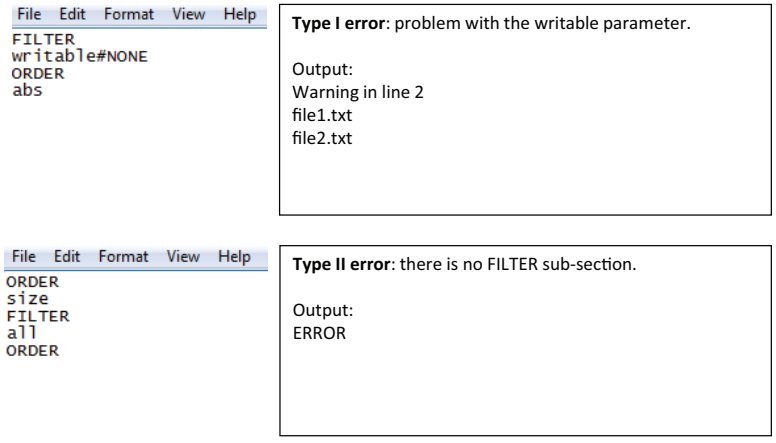

Figure 3: Example for the two types of errors. In the first example the program prints warning caused by an illegal *writable* parameter. Then, it prints the matched files (all files in this case, since the filter is illegal). In the second example the program simply prints ERROR without printing matched 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 on its own.
2. Warnings should be printed in the same order as they appear.
3. Printing the matched files should be done **after** printing warnings (type I errors) from all subsection. In case of multiple section in the **Command 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/, the file /abc/my#dir&1/work.txt/ may be filtered.
3. No other process changes the files under the **Source Directory** while your program is running.
4. Other than the cases described above ("YES"/"NO" parameters and illegal values for the *between* filter), you may assume the parameters are valid in all cases. You may **not** assume that the file structure or FILTER/ORDER names are valid (as explained in section 6).
5. Regarding white spaces in the input: you may assume there are no redundant white spaces anywhere in the **Command File**. This includes trailing or preceding white spaces, filter lines with more than one space separating the different filters, etc. You may ignore such cases in your program, and handle them as you see fit. Similarly, you may also assume there will be no empty lines in the **Command File**.

7 Design

Your program should follow the design shown in Tirgul 10. If you choose to work with a different design, you must explain it in details in your README file. In your explanation, you should specify why you chose that design, why it is preferable over 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. Describe the design of your project.
2. How did you implement your design? (Whether the one shown in class or your own design).
3. How did you handle the different errors?
4. which data structure did you used to sort your matched files?

8.2 Jar File

You should submit a file named ex6.jar containing all the *.java* files of your program, as well as the README file. Please note the following:

- Files should be submitted along with 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 accept your program and produce documentation.

You may use the following unix command to create the jar files:

*jar –cvf ex6.jar README oop/ex6/filescript/*java oop/ex6/orders/*java*

This command should be run from the main project directory (that is, where the *oop* directory is found).

9 Misc.

9.1 School Solution

A school solution can be found in ∼oop/bin/ex6SchoolSolution. 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 in the school solution. However, if you see some clearly unintended behavior from 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

A file called ex6_filters.zip is found in the course website. The zip file has 2 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.

Also, a file called ex6_source_directories.zip is found, with 2 corresponding directories:

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 submitting your exercise, we will run all the automatic tests presented above, and you will receive an email with the tests you failed on. Error messages will contain the filter number. For example, if you get the following error:

runTests[27](Ex6Tester): 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 when tested with the filter basic_filters/filter028.flt on basic_source_directory/. In order to fix such bugs, you can run the school solution against the detailed example and compare the output with yours. We will compare your program's output against the school solution on each filter in these sets (i.e., basic and advanced) and on unseen tests that will have a weight of 10% from the automatic test part. For your convenience, you may (and are advised to!) run the automatic tests from the terminal by running the following command:

~oop/bin/ex6 ex6.jar

(where ex6.jar is your jar file)

Good-Luck