CLIDS 2013 - exercise 2 – File processing tool

Goals: to exercise, experience

- Working according to a given design
- Working with (reading and copying) file contents and file attributes
- Working with directories and directory structures
- Working with the Exceptions mechanism

Submission Deadline: Thursday, 2/5/2013, 23:55

This exercise should be done **in pairs**. If you choose to solve it alone, please send us an email (clids@cs) about it ASAP. Please note that if many people wish to work alone we might pair people together.

Programming Environment

In this exercise you may use classes available under packages java.util.*, java.lang.*, java.io.* and java.text.* in standard java 1.7 distribution (as introduced in the intro2cs course). If you wish, you may also use classes from the java.**n**io package (although it is your responsibility to carefully read the API of this package and use it correctly). Apart from that, you are not allowed to use classes from any other packages. You are advised to use the **java.io.File** class for reading directories, the **java.text.SimpleDateFormat** for recognizing dates, **String** for working with text**,** and **java.util.TreeSet**, **java.util.Iterator** and **java.util.Comparator** for sorting and iterating data structures. You are highly encouraged to examine the full API of each class you use.

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 perform a set of **actions** on each of these files. It will also allow a user to define an **order** on the filtered files, such that the actions will be performed in this order.

Specifically, in this exercise you will implement a program, called **MyFileScript**, which is invoked from the command line as follows:

java clids.ex2.filescript.MyFileScript sourcedir commandfile

Where:

a. **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).

b. **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 triplets of FILTER/ACTION/ORDER subsections. The FILTER subsection includes filters which are used to select a subset of the files. The ACTION subsection includes a set of actions to be invoked on each of the filtered files. The ORDER subsection indicates in which order files should be traversed when performing action(s). Each subsection may include comments (lines starting with the "@" symbol).

Commands File structure

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

1) FILTER
2) ACTION
3) ORDER

The FILTER subsection

This subsection describes the filters that will be used in this section. Filters will **recursively** 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 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
before	File modified before (including) the *given* date	*dd/MM/yyyy*	before%21/02/2012
after	File modified after (including) the *given* date	*dd/MM/yyyy*	after%21/02/2012
greater_than	File size is more than the *given* number of k-bytes.	*double*	greater_than%5
between	File size is between 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
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*	prefix%.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

Comments:

1. You may assume filter format is always NAME%VALUE or NAME%VALUE%VALUE in the case of *between* filter (with no other suffix).
2. Date filters *(before* and *after*) receive values in the form dd/MM/yyyy, e.g. 21/02/2013. You are advised to parse it with *java.text.SimpleDateFormat*[1]. You may assume **dates are legal**.
3. The domain for size filters (*smaller_than*, *between* and *greater_than*) is any non-negative double number (java *double*, ≥ 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 the **validity of the input**.

[1] In *java.text.SimpleDateFormat*, **d** stands for day, **M** stands for month and **y** stand for year, while **m** stands for minute, so dd/MM/yyyy is the format you need to work with (and **not** dd/**mm**/yyyy).

4. *between* filter receives 2 values separated by %. For example, *between*%10%20 should return all files with 10<=size<=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 Error Handling section).
5. Conversion between bytes and k-bytes is straightforward: 1 kb = 1024 bytes.
6. For the *writable/executable/hidden* filters, the domain is YES/NO strings. You **need to verify this**. Other values are considered errors
7. For *file*, *prefix* and *suffix* 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. *prefix/suffix* filters match file names that start/end with the filter value. 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").
8. You may assume that there is at most one filter after the FILTER line.

A few more things:

1. A FILTER sub-section must appear in every section. Otherwise it is an error.
2. A FILTER sub-section may be empty **(i.e., the** FILTER line with **no other lines**). In that case, all files in the directory are matched.
3. 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).
4. 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").
5. Lines starting with '@' are considered comments and should not be treated as filter lines. Zero or more comments may come at any point **after the FILTER line** (i.e., before or after the actual filter). This means that the first line of the file may not be a comment (this is considered an error).

The ACTION subsection

Action subsections always include one or more action lines of the form: **ACTION%PARAMETER**.
An ACTION subsection always comes after a FILTER subsection. Each action in this subsection should be performed on all files that are satisfied by the filters **from the previous subsection**. The following are possible actions:

Action	Meaning	Parameter	Example
print_data	Print file information to the standard input (STDOUT)	–	print_data
print_name	Print file name to the standard input (STDOUT)	–	print_name
copy	Copy file to the given target directory	*Target directory*	copy%dir1/dir2/
exec	Set file executing permissions	*YES* or *NO*	exec%YES
write	Set file writing permissions	*YES* or *NO*	write%YES
last_mod	Set file's last modified time	*dd/MM/yyyy*	last_mod%21/02/2013

1. Action string format is **ACTION** (print actions) or **ACTION%PARAMETER** (with no other suffix). You are not required to validate this.
2. Actions are performed only on files (not on directories).

3. You may assume each line contains only a single action (and not more). However, ACTION sub-sections may include multiple actions.
4. *print_data*: print one line containing the following information about the file:
    - 3 consecutive characters indicating:
        - is it hidden ("h") or not ("-")
        - is it writable ("w") or not ("-")
        - is it executable ("x") or not ("-")
    - a white space
    - file size (in k-bytes, a java *double* variable)
    - another white space
    - the file's last modification date (as returned by Date.toString())
    - another white space
    - the full file path (using getAbsolutePath ())
    - A new line character

    For example:

    A non-hidden, executable file (but not writable) of size 0.0859375 k-bytes, which was last modified on 29/02/2011 on 13:51:15, called OkFile:

    --x 0.0859375 Thu Dec 29 13:51:15 IST 2011 /cs/course/current/clids/tmp/OkFile
5. *print_name*: print file name (excluding path), and a new line. For example:
   OkFile
6. *copy*: Target directory is always relative to the **Source Directory**. A target directory may **not** be absolute (i.e., it must not start with "/" (this is considered an error)).
    - A filtered file is **not** copied along with its directory structures. For example, for **Source Directory** **/aaa/** and target directory **bbb/ddd/eee/**, a filtered file **/aaa/abc/def/work.txt** should be copied to **/aaa/bbb/ddd/eee/work.txt** (without the **abc/def/** sub-directories)
    - If the target directory doesn't exist, it should be created.
        - You may assume the target directory is not an existing file, and does not contain any existing file along its path. For example, in the example above (**bbb/ddd/eee/**), neither **bbb**, nor **bbb/ddd**, nor **bbb/ddd/eee** are existing files (these directories may exist though).
        - Furthermore, if the target directory exists, and contains some file (say *foo.txt*), you may assume you will not copy other files with the same name into this directory.
        - As a result, you may assume that no 2 files with the same name are copied to the same directory.
        - To conclude, there is no need to check the validity of the input before performing the copy operation (except for the "/" test presented above).
    - **File**.*mkdirs*() and **File**. getAbsolutePath() may be useful when implementing this action.
    - When copying a file, there is no need to preserve the old file's permissions (it is ok if they are different than the original permissions).
    - Unless you choose to use java 7's java.nio package (see java 7 section), you are required implement the copy operation on your own, using the methods studied in class, as there is no general java command for copying files in earlier java versions.
7. For actions that change the file's permissions (*exec* & *write*):
    - Permissions should be changed only for the current user (not *group* permissions or *all* permissions). This is compatible with the *java.io.File* class methods.

- You must validate that parameter values are either "YES" or "NO". Other values are considered errors.

8. All actions (except *print* actions) may fail due to permission problems. Such cases are errors.
9. An ACTION sub-section may be empty (i.e., the ACTION line with no other lines). In that case, you should not perform any action.
10. Actions are traversed by the order in which they appear in the **Commands File**. For each action, all files are traversed (according to the ORDER subsection. See below), and actions are performed. That is, we perform the first action on all files, then the second action on all files, etc.
11. There is no need to perform any action on copied files in the current section. That is, say your FILTER subsection filters a single file called *file.txt.* Then, your ACTION subsection contains the *copy%dir1* action, followed by the *print_name* action. In such case, you first copy *file.txt* to the directory *dir1*. Now, you have 2 files called *file.txt* (the original file and the new, copied file). However, *print_name* should only be performed on the original file (i.e., print *file.txt* once and not twice).
12. In case there are multiple sections, actions **are** performed on copied files (if there any) **from previous sections**. That is, in the previous example, if the *print_name* command is part of one of the next sections, it should print *file.txt* **twice**.
13. Comments lines (lines starting with '@') are allowed anywhere inside this subsection.

The ORDER subsection

This subsection indicates the order in which filtered files are traversed when actions are performed. The following are possible orders:

Order	Meaning
abs	Sort files by absolute name (using getAbsolutePath()), going from 'a' to 'z'.
file	Sort files by file name (excluding directory), going from 'a' to 'z'.
mod	Sort files by modification date, going from oldest to newest.
size	Sort files by file size, going from smallest to largest.

1. You may use **java.util.TreeSet** and **java.util.Comparator** to sort your files. Please carefully review the API's of these classes before doing so. You may also implement your own data structures and order mechanisms.
2. For string orders (*abs* and *file*), use the *String.compareTo*() method to compare 2 file names.
3. In every section in the **Commands File**, the ORDER subsection is optional. That it, it may or may not appear in this section. In case it does not appear, the *abs* order should be used. This also applies if the ORDER line appears without any specific order (i.e., it is followed by the next FILTER subsection or by the end of the **Commands File**).
4. You may assume every order subsection contains at most a single (and not more) order line.
5. In the *mod* order, two files are considered equal if they were last modified on the same hour, even if on different minutes.
6. In case two or more files are equal according to any of the *file*, *mod* or *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.
7. In the *abs* order, capital letters will be ordered after the lowercase letters. (e.g., a.txt should be printed before A.txt).

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 potential errors, divided to three sections.

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.

The errors in the third section (type III errors), are errors that cannot be detected in advance. You should print a warning message when you encounter the problem and continue normally.

Type I errors

1. A bad FILTER/ACTION/ORDER name (e.g., *greaaaater_than*). These names are also **case-sensitive** (e.g., *Mod* is an **illegal** order name, and should result in an error).
2. Bad parameters (anything other than YES/NO) to the *hidden/writable/executable* filters, or the *write/exec* actions.
3. Illegal values for the *between* filter (*between*%15%7).
4. A destination argument for the *copy* action that starts with '/'.

Comments:

1. Type I errors should result in printing "Warning in line X" and continuing normally, skipping these errors (X is the line number where the ACTION/FILTER problem occurred). All warnings are printed together, before printing comments and performing any action.
2. In case there is one filter/action and there is a warning which causes the sub-sections to be empty, you should behave as if there is no filter/action there (i.e., for an empty filter – you should match all files and for empty action – do nothing).

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. The first line in the file being a comment.
8. A bad subsection name (i.e., not FILTER/ACTION/ORDER). Subsection names are **case-sensitive** (e.g., *filter* is an **illegal** subsection name, and should result in an error).
9. Bad order of **Commands File**. E.g., no ACTION subsection.

Comments:

1. Type II errors should result in printing "ERROR" (with newline) to STDERR (using *System.err*) and exiting the program (use *System.exit*(-1)). You are allowed to let the main file catch all those thrown exceptions and handle them there.
2. Upon any error in the **Commands File** (errors 6-10), your program should not **perform any action,** or **print any comment or warning** (type I errors). This includes a file with 2 sections, where the first section is proper and the second isn't.

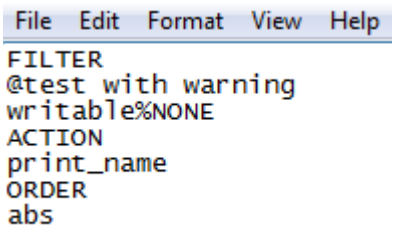
Type III errors:

Action failure (e.g., trying to copy a file without reading permission).

Comments:

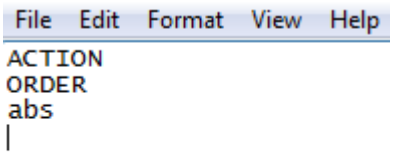
Type III errors should result in printing "Action failed in line X" when the error occurred. (X is the line number), skip that action and continue normally.

Examples:



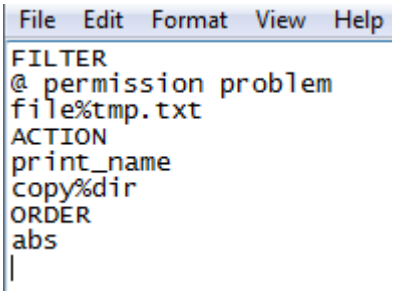
Type I error: problem with the writable parameter.

Output:
Warning in line 3
@test with warning
File1.txt
File2.txt



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

Output:
ERROR



Type III error: No reading permission

Output:
@ permission problem
tmp.txt
Action failed in line 6

In the first example the program first prints the warning and the comments. Then, it performs the action (*print_name*) on the files matched by the filter (all files in this case, since the filter is illegal). In the second example the program simply prints ERROR without performing any action. In the third example the file tmp.txt doesn't have reading permission so the program prints an error message once it reaches this line.

General remarks

1. Recall that multiple FILTER+ACTION+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. Regarding comments and warnings:

a. Comments ('@' lines) may appear in any of the sections, except before the first FILTER subsection.Every comment, warning and action failure should be printed to the standard output (*System.out*). EXIT (type II errors) should be printed to stderr (*System.err*).
b. Printing comments and warnings (type I errors) should be done **before** actions are performed: Warnings (from all subsections) first, then comments (from all subsection).
c. If the **Commands file** contains more than one section, each section's comments and warnings should be printed before performing the actions in that section. That is, the order should be:

Print warning of section 1
Print comments of section 1
Perform actions of section 1
Print warning of section 2
Print comments of section 2
Perform actions of section 2

...

d. Comments and warnings should be printed in the same order as they appear.

3. As noted above, when performing the actions you first traverse the actions, and perform each action on all the files.
4. Error type III will be printed during the "*Perform actions of section X*" part above because they cannot be identified in advance.

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:

- The file system (under the **Source Directory**) is a real tree (no hard or symbolic links).
- 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.
- No other process changes the files under the **Source Directory** while your program is running.
- ~~A filter expression may be contradictory, i.e. it may not be satisfied by any file. For example, a 2-line filter composed of the lines "writable%YES" and "writable%NO" cannot be satisfied by any file. For simplicity, you do not need to check whether a filter expression is contradictory and treat such cases differently. The list of files in this case will be empty, which is acceptable (an empty set of files may be obtained also for non-contradictory filters).~~
- Other than the cases described above ("YES"/"NO" parameters, a copy destination directory starting with "/" 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/ACTION/ORDER names are valid (invalid names are considered an error).
- Regarding white spaces in the input: you may assume there will be 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**.

Java 7

Version 7 of Java introduces a new Input/Output package (java.**n**io) as we briefly saw in class. This package includes several operations not included in the java.io package (for example, a copy method). In this exercise you are allowed to use this package. If you do so, we encourage you to read the java.nio API carefully. You should explain in detail the part of your code that uses this package.

Design

Your program should follow the design shown in lecture 6. 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 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.

README

As in ex1, you are required to hand in a README file.

Other than the standard README sections described in the ex1 description, your README file should explain the following issues:

1) How did you implement your design? (Whether the one shown in class or your own design)
2) How did you handle the different errors?
3) Explain your choice between an abstract or interface classes for the filters, actions and orders (see lecture 6, slide 14).

Advices

1. Develop and test each class separately, before using it with other classes. For that, each of your classes should include a short main that tests the functionality of the class. Please remember to remove these *main* methods before submitting your code.
2. We suggest you test your FILTER and ORDER subsection with the *print_name* or *print_data* commands, as they are easiest to debug. Similarly, check your ACTION subsection with a simple filter and a few files. After you are certain that each part works well on its own, start performing more complicated tests.

Submission Guidelines 

Submit a file named ex2.jar containing all the *.java* files of your program, as well as the README file. Files should be submitted along with their original packages. It should not contain any *.class* files. Remember that you program must compile without any errors or warnings, and that *javadoc* should accept your program and produce documentation.

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

*jar –cvf ex2.jar README clids/ex2/filescript/*java clids/ex2/actions/*java …*

Automatic testing

A file called ex2_filters.zip is found in the course website. The zip file has 4 folders, each containing filter files:

a) basic_filters/ - *simple tests of the filters and orders.*
b) advanced_filters/ - *more advanced testing of the filters and orders.*

c) complex_filters/ - *testing your code against complex filters and directory trees*.
d) actions_filters/ - *testing your actions*.

Also, a file called ex2_source_directories.zip is found, with 4 corresponding directories:

a) basic_source_directory/
b) advanced_source_directory/
c) complex_source_directory/
d) actions_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/*.

As introduced in ex1, 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, you may receive the following error:

```
### Command line error, when ran with arguments basic_source_directory/
basic_filters/filter006.flt : problem with handling a file with executable%NO
filter
```

This means that your program generated an output different than the school solution when tested with the filter `basic_filters/filter006.flt` on `basic_source_directory/`. What comes after the ':' sign is a short description of the error (in the example above, your program had a problem with handling the executable%NO filter). In order to fix such bugs, you could run your program against the detailed example, run the school solution against it, and compare the results, and see where your problem is.
The first 3 filter sets (i.e., *basic*, *advanced* and *complex*) only contain printing actions and will thus only compare your program's output against the school solution. The last filter set (*actions*) tests other actions.

Important note: as in previous exercises, when you submit the exercise via the course web site, your exercise is tested automatically. When many students submit at the same time, a delay of the report mail is expected. Instead of re-submitting the exercise every time, you can run an external script that generates the **same results**. Running this script BEFORE submitting the exercise via the website (and waiting for an email each time) is a much better alternative.
In order to do that, simply run the following command on your terminal (ex2.jar is your jar file):

~clids/bin/ex2AutoTests.sh ex2.jar

Grading

- Working according to Coding style guidelines - 10%
- Design explanation (1+2 in README section)– 30%
- Answering a question (3 in README section) – 10%
- Correct implementation (automatic tests) - 50%

If you submit a file that fails some of the automatic tests, you will lose a number of points proportional to the number of failed tests and the type of tests you failed. For example, failing many tests due to the same reason (i.e., one bug) will result in losing less points than failing many tests due to different reasons (i.e., many bugs).

School solution:
A school solution can be found in: *~clids/bin/ex2SchoolSolution*

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.

Forums and questions
An ex2 discussion forum is available in the course website. We will review it every 2-3 days and answer the relevant questions. In order to make the best of the forum, we advise you, before posting questions in the forum, to review both the exercise definition and the previously asked questions carefully, as well as experiment with the school solution. The *search* option in the forum may come in handy for querying previously asked questions. Asking previously asked questions will result in (a) slower response time on our behalf (due to irrelevant questions) and (b) difficulties on the students' behalf to extract the relevant issues from the forum.

Important clarifications will be posted in the official announcements forum. Messages in this forum are **obligatory**.