Line Filter

1 Introduction

This project gives you practice with

- console input/output,
- using command line arguments,
- using the `java.util.Scanner` class,
- `String` methods,
- conditional logic,
- simple loops,
- redirecting `stdin` and `stdout`, and
- using the Unix "pipe and filter" paradigm.

2 Problem Description

You want to see which lines – possibly in a file – contain a given string of characters.

3 Solution Description

Write a Java class that is executable from the command line named `LineFilter`. `LineFilter` should read lines from `stdin` (`System.in`) and print to `stdout` (`System.out`) any line that `contains` the text given in the first command line argument. If no command line argument is given, it should print every line.

3.1 Basic Usage

When you run your program it should give no prompt and wait for the user to enter a line of text and hit enter. If you give no command line arguments when you run your program it should simply echo the user's input on the next line, then wait for the user to enter another line. You can stop the program by entering Ctrl-D. Ctrl-D signals to the terminal that the input stream will produce no more data, and will cause `Scanner`'s `hasNext` method to return `false`.

```
$ java LineFilter
Hello.
Hello. # Printed by program
Who are you?
Who are you? # Printed by program
Stop copying me!
Stop copying me! # Printed by program
I give up.
I give up. # Printed by program
$
```

When you run your program with a command line argument, it only echoes the user's input if it contains the text in the command line argument.

```
$ java LineFilter Simon
Touch your head.
Simon says touch your head.
Simon says touch your head. # Printed by program
Jump up and down.
simon says jump up and down.
Simon says jump up and down.
Simon says jump up and down. # Printed by program
$
```

3.2 Pipe and Filter

Note: this section is not strictly necessary for this homework; we provide this for your edification. Some of the command line utilities we discuss here may not be available on Windows by default. You can look for equivalents in Windows PowerShell, install GOW (GNU on Windows) or UnixUtils , or run Cygwin ; but the best idea is to install and use Unix. Unix has a learning curve, but you will thank yourself a thousand times over for learning it.

Your program reads input from `stdin` (`System.in`) and writes output to `stdout` (`System.out`). In most command shells you can redirect `stdin` and `stdout`. The right angle bracket character, >, redirects `stdout`, and the left angle bracket character, <, redirects `stdin` First, try running your program with a command line argument but redirect output to a file.

```
$ java LineFilter Simon > simon-answers.txt
Touch your head.
Simon says touch your head.
Jump up and down.
Simon says jump up and down.
$
```

Notice that the program didn't print anything to the console. That's because we redirected `stdout` (`System.out`) to a file. You can see the output by printing the contents of `simon-answers.txt` with `cat` (`type` on Windows).

```
$ cat simon-answers.txt
Simon says touch your head.
Simon says jump up and down.
```

Now try redirecting the input to your program. We'll use the file we just created, `simon-output.txt`, for input. Notice that the program doesn't prompt the user for input and prints all the lines from `simon-input.txt` (because they each contain the text `Simon`).

```
$ java LineFilter Simon < simon-answers.txt
Simon says touch your head.
Simon says jump up and down.
```

Unix contains many utilities that process text, and these utilities can be combined by piping the output of programs that produce text through other programs that read text and produce text. We do this by "piping" the output of one program to the input of another program using the pipe operator, `|`. The pipe operator redirects the output of the program on it's left to the input of the program on its right, and redirects the input of the program on its right to the output of the program on its left. You can think of the text as flowing from left to right on the command line.

Here's a simple example. The `sort` utility reads lines from `stdin` and prints them in sorted order. Run `sort` and enter some lines of text, followed by Ctrl-D.

```
$ sort
zz top
led zepelin
abba
ace of base
# Press Ctrl-D
abba
ace of base
led zepelin
zz top
$
```

Works much like your `LineFilter` program, doesn't it? Using a utility like `sort` by itself isn't interesting. Utilities like `sort` are most useful as *filters*. Now try piping the output of `cat simon-answers.txt` through `sort`:

```
$ cat simon-answers.txt | sort
Simon says jump up and down.
Simon says touch your head.
```

You can also redirect `sort`'s output to a file.

```
$ cat simon-answers.txt | sort > simon-answers-sorted.txt
$ cat simon-answers-sorted.txt
Simon says jump up and down.
Simon says touch your head.
```

And you can pipe through your `LineFilter` program as well.

```
$ cat LineFilter.java | java LineFilter public
public class LineFilter {
   public static void main(String[] args) {
$
```

You can also pipe through multiple stages for more detailed processing. Here we get a sorted list of famous hackers named John.

```
$ cat famous-hackers.txt | java LineFilter John | sort
John Backus
John Carmack
John Kemeny
John McCarthy
John Ousterhout
$
```

For a more detailed discussion of `stdin`, `stdout` and `stderr`, see this page .

4 Checkstyle

Review the CS 1331 Code Conventions and download the Checkstyle jar file and the configuration file to the directory that contains your Java source files. Run Checkstyle on your code like this (in the directory containing all your Java source files):

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count the errors by piping the output of Checkstyle through `grep`.

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | grep -cEv "(Starting
    audit...|Audit done)"
0
$
```

The `-c` option tells `grep` to count matching lines instead of printing them, `E` means use `egrep` (extended `grep`) syntax, and `v` means invert the match. Here we use an inverted match to discard the two non-error lines of Checkstyle's output. If you have `egrep` you could leave off the `-E` and just use `egrep`.

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | egrep -cv "(Starting
    audit...|Audit done)"
0
$
```

In future homework projects we will run Checkstyle on all the Java source files you submit and deduct one point from your score for each style error found by Checkstyle.

5 Turn-in Procedure

Submit your `LineFilter.java` file on T-Square as an attachment. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double-check that you have submitted and not just saved a draft.

6 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
   (c) Helps find last minute causes of files not compiling and/or running.

[1]Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight Friday. Do not wait until the last minute!