$Id: asg1j-jfmt-filesargs.mm,v 1.10 2013-09-25 10:51:19-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/asg1j-jfmt-filesargs
URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Assignments/asg1j-jfmt-filesargs/

1. Overview

In this assignment, you will learn how to access files, and make use of command line arguments by writing a utility similar to the program **fmt**(1). Read the man page for **fmt** by typeing the command:

```
man -s 1 fmt
```

Whenever you see the name of a command in bold face followed by a number in parentheses, it refers to a command in a certain section of the Unix man pages. In this case, section 1 contains the documentation for the program **fmt**. Your program will be similar but not identical. Follow the specification given here. There is also a reference implementation written in Perl (**misc/pfmt.perl**).

2. Program Specification

We present the program specification in the form of a Unix **man**(1) page.

NAME

jfmt — simple text formatter

SYNOPSIS

jfmt [-*width*] [*filename* . . .]

DESCRIPTION

The format utility reads in text lines from all of its input files and writes its output to stdout. Error messages are written to stderr. Each file is handled separately, and within each file, all consecutive sequences of lines containing non-whitespace characters are considered as a single paragraph. A paragraph is written out with a maximal sequence of words not to exceed the specified output line width. It is then followed by one empty line.

OPTIONS

-*width*

The specified width is the maximum number of characters in an output line. If a word that is longer than width is found, it is printed on a line by itself. The default width is 65 characters.

OPERANDS

Each operand is a file name, and they are read in sequence. An option is recognized as such only if it begins with a minus sign and precedes all operands. If any operand is specified as a single minus sign (-), then stdin is read at that point.

EXIT STATUS

0 All files were read successfully, output was sucessfully generated, and no errors were detected.

1 Errors were detected and messages were written to stderr.

SEE ALSO

fmt(1), **pfmt.perl**.

3. Implementation Sequence

Whenever your write a program that is non-trivial, you should follow some iplementation sequence that develops a program a little bit at a time, keeping a working program and adding to it little by little. Make backups frequently.

(1) Start with the given code: The **code/** subdirectory contains some starter code for yor project. The **misc/** subdirectory contains some examples you should study, but which are not directly useable in your code.

(2) Make sure `checksource` and `cid` are in your `$PATH` environment variable. The first checks on basic formatting, and the second will check your code into an `RCS` archive.

(3) Play around with `pfmt.perl` to see what your program should do. If you copy this file into your development directory, make sure the `x`-bit is on.

(4) The program `jfmt.java`, as given, merely cycles through the files and prints debug code. Each line is printed as it is read, and each word within that line is read. Note that words are selected from a line with a white space regular expression passed to `split`. `"\\s+"` is a string representing the regular expression `\s+`, which matches any sequence of white space.

(5) As you are developing Java code, you should have your browser pointed at
`http://docs.oracle.com/javase/7/docs/api/index.html`,
which is the Java™ Platform, Standard Edition 7 API Specification.

(6) Your program should have the following import statements at the beginning:
```
import java.io.*;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import static java.lang.System.*;
```
Important note: In this assignment, you may use anything in `java.util`, but in general in this course you are prohibited from using anything from that package except when explicitly authorized. See the syllabus for a list of classes in `java.util` that you may always use.

(7) Read about interface `java.util.List` and class `java.util.LinkedList`. Then, at the beginning of the function `format`, add in the following declaration:
```
List<String> words = new LinkedList<String>();
```
Delete the line that echos input lines and instead of printing the words found in each line, `add` them to the list of words. Note that you need to take special care not to add an empty leading word to the list of words. (What does `split` do when there is a space ahead of the first word?)

(8) Create a function `print_paragraph` which is called whenever you see the end of a paragraph. This function removes and prints each word in the list. The end of a paragraph occurs at end of file. It also occurs with an empty line, i.e., a line that contains only spaces and tabs or nothing at all. Sequences of empty lines produce the same output as a single empty line.

(9) Look at interface `java.util.Iterator`. The function `print_paragraph` should create an iterator over the list:
```
for (String word: words)
```
As you iterate over each word, print it, with a space between words, but not before the first word, nor after the last word of the line. Whenever printing a word would exceed the line length, print a newline character. Note that the first word on a line is printed whether or not it exceeds the line length.

(10) When you are finished with the list, empty it:
```
words.clear();
```

(11) An alternative method is to create a `StringBuffer` to accumulate words until the line is full. In this case, don't use string concatenation (`+`). Use `StringBuffer.append`. This is what the inner loop of the Perl code does. Study it an duplicate its functionality. Use `diff`(1) to check the output of your program against that of `pfmt.perl`.

(12) Add code to examine `args` to see if there is a maximum line length parameter, and if so, keep it in a variable. If `args[0]` exists, it is a valid option if `args[0].matches ("-\\d+")`, an invalid option if `args[0].matches ("-.+")`, and otherwise not an option at all.

(13) A string may be converted to an integer by using `Integer.parseInt` (see the Java 1.5 API documentation for `java.lang.Integer`). You can either use `substring` to ignore the minus sign, or negate the result after conversion. Note that any word from the command line that `equals ("-")` is not an option, but an operand. Instead of bothering to check for formatting

using the `matches` function in the previous point, You could just assume it is valid, but use a `try-catch` construct to catch a `NumberFormatException` if not.

4. Program Testing

It is extremely important to test your program and verify that it works under all possible circumstances. The subdirectory `.score` contains some files: `SCORE` is the grader's score sheet, to be used to evaluate your program. After submitting your code, run `testsubmit` and then `cd` into the directory. Copy the files `mk.build` and `mk.test` and run them to verify that your program works.

To see exactly what testing should look like, copy the input files into your baseline directory (separate from your development directory), make a symbolic link to make `jfmt` point at `pfmt.perl` and see what output the Perl program produces. Your output should be identical, including `stdin`, `stdout`, and the exit status.

5. What to Submit

Submit the files `README`, `Makefile`, `jfmt.java`. If you are doing pair programming, be sure to edit your `README`, as required in the pair programming directory, and add the `PARTNER` file to your submit list.

Verify the submit! This is explained in `lab1`. If you don't submit all of the necessary code, you will lose many points. Do not submit the jar file or any class files. Make sure your `Makefile` works. Do a pre-grade using the information in the `.score` subdirectory.