UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2013

Programming Project #1: A Text Formatter (Version 5)

Due: Monday, 21 October 2013 at midnight

The TeX program (and LaTeX, which is built on it) format raw text into neatly formatted lines grouped into paragraphs and pages. Unlike WYSIWYG[1] systems like Microsoft Word, these programs use plain ASCII text as their input, and define certain escape sequences that allow authors to *mark up* the text with commands that control the formatting. In this project, you'll implement a very simple version of these programs, called `tex61`. The description in the following sections may look complicated, but mainly this is just the result of being very specific about the desired result, including fussy little corner cases. For the most part, the gist of the specification is "format text as you'd expect."

1 Executing the Program

The command

```
java tex61.Main INPUTFILE.tx
```

should run your program on the given file and produce output on the standard output. The command

```
java tex61.Main INPUTFILE.tx OUTPUTFILE
```

should run the program and write output to the indicated file. The skeleton is actually all set up to do this. Examine and understand it, since we may expect you to do more of this setup in future projects.

2 Input Processing

Apart from commands, the input may be considered to consist of a sequence of words, paragraph ends, intervening horizontal white space (blanks and tabs), and line terminators ("newlines" or "linefeeds"). For this purpose, a "word" is a sequence of printable, non-blank characters. A "paragraph end" is either a null line—one or more adjacent lines containing only whitespace—or an end-of-file (possibly preceded by null lines). White space at the beginning or end of a line is ignored, as are paragraph ends at the beginning of the file. Any stretch of white space between words is equivalent to a single blank. Newlines that separate non-blank lines also act like blanks when filling text (see below).

[1]That is, "What You See Is What You Get."

The main activity of the program is to format text by *filling* and *justifying* it. To *fill* is to move words from line to line so as to fill each line as closely as possible to some maximum length, called the *text width.* To *justify* is to insert spaces more-or-less uniformly between words on a line so as to make all lines the same length, except possibly the last in each paragraph.

Filling. At the beginning of each line containing text, there is an amount of white space known as the *indentation.* The first line of each paragraph is to be indented by an additional amount known as the *paragraph indentation.*

If it is impossible to make a line fit in the text width (because it consists of a huge word that, together with paragraph indentation, is longer than the text width), simply output that word with the usual indentation, even though it will stick out at the right.

Justifying. To justify text, we insert spaces between words in an attempt to make the total length of the text on a line (plus indentation) equal to the text width. The last line of a paragraph and a line consisting of a single word are not justified—all their words are separated by single blanks. We only justify when filling; when we stop filling (by using the `\nofill` command), we also stop justifying until the next `\fill` command.

We only justify lines with $N > 1$ words. Suppose the text width is W, the line has P spaces of indentation, and the total number of characters in all the words on the line is L. We assume that filling guarantees that $B = W - L - P \geq N - 1$ when $N > 1$—that is, that there is enough space on the line to put at least one blank between words. Number the words on the line from 0 to $N - 1$.

1. If $B \geq 3(N-1)$, then insert 3 blanks between each pair of words. This finishes the line (it may be short, but we never put more than 3 blanks between words to prevent ridiculous-looking spacing.)

2. Otherwise, distribute spaces more-or-less evenly between words by arranging that the total number of blanks inserted between words 0 through k $(0 < k \leq N - 1)$ is always $\lfloor 0.5 + Bk/(N-1) \rfloor$.

For example, if $W = 72$, $P = 3$, and the words on our line are

```
The following quotation about writing test programs for a document
```

Then $N = 10$, $L = 57$, and $B = 12$. When justified, the line will be

```
␣␣␣The␣following␣␣quotation␣about␣writing␣␣test␣programs␣for␣␣a␣document
```

So we have inserted $\lfloor 0.5 + 12 \cdot 1/9 \rfloor = 1$ blanks between words 0 and 1, $\lfloor 0.5 + 12 \cdot 2/9 \rfloor = 3$ in total between words 0 through 2, $\lfloor 0.5 + 12 \cdot 4/9 \rfloor = 5$ between words 0 through 4, and $\lfloor 0.5 + 12 \cdot 5/9 \rfloor = 7$ in total between words 0 through 5.

Pagination. The formatter starts a new page whenever it has to output a line, but the current one is full (its number of lines is equal to the "text height"). The standard character for indicating a new page is the form feed (a *control character* whose ASCII encoding is 12, and is denoted in Java string literals by the escape sequence '`\f`'). It goes at the beginning of the first line of each page other than the first. If the line that won't fit on a given page is blank, then it and any other blank lines following it are discarded, and the new page starts with the first non-blank line.

3 Commands

The input text may contain *commands,* which generally modify the current formatting parameters. Table 1 shows the available commands. A command comes in one of two forms—parameterless:

\commandname

and parametrized:

\commandname{*balanced text*}

Here, *balanced text* means text that contains no bare curly braces (`{}`) or in which every instance of a bare left curly brace can be paired up one-to-one with a subsequent bare right curly brace. Here, "bare" means without a leading escape character, as in `\{` or `\}`. For example, the following are all balanced:

```
\indent{0}
\endnote{See Knuth, Digital Typography.}
\endnote{\parindent{2}Jan Tschichold, translated by Hajo Hadeler.}
\endnote{For begin, you can use left curly brace (\{)}
```

The characters '\\', '{' and '}' may only appear as provided in the commands below.

As it happens, only the `\endnote` command can have non-numeric text in it, and endnotes may not be nested in endnotes (apologies to Terry Pratchett), so you'll only have to deal with braces nested one deep inside of other braces.

Except as otherwise noted, commands in general generate no text (they are replaced by empty strings). Certain commands (such as `\textwidth`) change formatting parameters. The points at which they take effect when used in the middle of a line (or page, for commands like `\textheight`) are undefined.

Figure 1 gives an example of possible input and the resulting output.

4 Endnotes

The \endnote command creates a numbered endnote, leaving behind a square-bracketed reference (e.g., "[1]"). The text of all endnotes goes at the end of the document, immediately after its last line (i.e., not preceded by a paragraph skip). The text of each endnote starts with the reference number in a new paragraph (as if preceded by a blank line), in the same format as the reference in the text. Follow the reference number with the equivalent of '\␣' (backslash blank) so that the first word of the note is separated from the reference by a single space. That is, treat the reference and following space as part of the first word, so that justification does not add extra space between them. Number endnotes consecutively throughout the document, starting at 1. See Figure 1 for examples.

The formatting parameters for endnotes are separate from those for regular text, and their initial values are different (see the table in §3.) The "`\textheight`" command is ignored. The text height used for the endnotes is whatever height is in effect at the end of the main text. The effect of other commands (like `\textwidth`) carries over from endnote to endnote and does not "leak" back into the main text (or vice-versa).

5 What to Turn In

You will need to turn in `tex61/Main.java`, `tex61/UnitTest.java` and any other source files you use in your solution.

Command	Effect	Default in main text	Default in endnotes
\indent{N}	Set the current indentation to N spaces, effective no later than the first line of the next paragraph that is output after processing this command.	0	4
\parindent{N}	Set the current paragraph indentation to N spaces, effective effective no later than the first line of the next paragraph that is output after processing this command. Subsequent paragraphs will be subject to this much indentation in addition to that set by \indent.	3	-4
\textwidth{N}	Set the current text width to N. A line is considered full (for purposes of filling) when the number of characters on it (including indentation) is equal to N. This command takes effect no later than the next line of output that is started after this command is processed.	72	72
\textheight{N}	Set the current text height to N. This is the maximum number of lines that may appear on a page before the next page starts (if the current page already has too many lines, a new page thus starts immediately). This takes effect no later than the next page of output that starts after the command is processed.	40	∞
\parskip{N}	Insert N blank lines in front of each subsequent paragraph. Blank lines that would go at the top of a page as a result are skipped. This command takes effect when (not before) the first line of the next paragraph is output after this command is processed. Thus, putting a `\parskip{3}` outputs three lines before the first line of the next paragraph whether the `\parskip` occurs before or after the blank line marking the end of the preceding paragraph.	1	0
\\\\	A backslash character.		
\\{	A left curly brace character.		
\\}	A right curly brace character.		
\␣	(Backslash followed by space) a character that prints as a space, but is counted as if it were a word character (like a letter). Thus, filling will not break '`number\␣1`' across lines.		
\nofill	Stop filling and justifying lines. Upon encountering a line (or paragraph) termination, output all accumulated text since the last output line, with words separated by single blanks, subject to the usual indentation. The text width is ignored. This takes effect no later than the start of the next paragraph.		
\fill	Fill lines as usual (and justify, if specified). This takes effect no later than the start of the next paragraph.		
\justify	Commence justifying lines whenever in fill mode.		
\nojustify	Stop justifying text (even when filling).		
\endnote{*TEXT*}	Suspend current accumulation of ordinary text and add a new endnote containing *TEXT* (see §4).		

Table 1: Mark-up commands for the text formatter.

```
The following quotation about writing test programs for
a document compiler from an article by Donald Knuth conveys some
of the right frame of mind\endnote{(D. E. Knuth, ``The Errors of TEX'',
Software Practice & Experience, 19 (7) (July, 1989), pp. 625-626).}:

\parindent{0}\textwidth{68}\indent{4}\parskip{0}
"... I generally
get best results by writing a test program that no sane user would ever think
of writing.... The resulting test program is so crazy that I could not
possibly explain to anybody else what it is supposed to do...."

\textwidth{72}\indent{0}\parindent{0}
You should also read Column 3 of _More
Programming Pearls_\endnote{J. Bentley, More Programming Pearls: Confessions of a Coder,
AT&T Bell Telephone Laboratories, 1988.}  for more advice.
Among other things, this column discusses scaffolding.

\parskip{1}\parindent{4}
The standard C header file assert.h provides a macro
"assert".  When this file is included in a source file, and the
file is compiled                       without
the option  -DNDEBUG, each call assert(C), where C is
any boolean expression, will terminate the program with an error
message if C turns out to be false at that point.

\parindent{0}
References
```

```
   The following  quotation about writing  test programs for  a document
compiler from an article by Donald Knuth conveys some of the right frame
of mind[1]:
    "... I generally get best results by writing a test program that
    no sane user would ever  think of writing.... The resulting test
    program is so crazy that I could not possibly explain to anybody
    else what it is supposed to do...."
You should also  read Column 3 of _More Programming  Pearls_[2] for more
advice. Among other things, this column discusses scaffolding.

    The standard C header file  assert.h provides a macro "assert". When
this file is included in a source file, and the file is compiled without
the  option  -DNDEBUG, each  call  assert(C),  where  C is  any  boolean
expression, will terminate the program with  an error message if C turns
out to be false at that point.

References
[1] (D. E. Knuth, ``The Errors of TEX'', Software Practice & Experience,
    19 (7) (July, 1989), pp. 625-626).
[2] J. Bentley,  More Programming Pearls:  Confessions of a  Coder, AT&T
    Bell Telephone Laboratories, 1988.
```

Figure 1: Example of input (above) and resulting output (below).

We will evaluate your project in part on the thoroughness of your testing. Put JUnit test classes with names ending in `Test` (e.g., `PageAssemblerTest.java`) in your `tex61` package, including one particular class named `UnitTest`, which should run *all* your individual JUnit tests as a *suite* (see the skeleton). Our autograder script will run `UnitTest` in the JUnit framework, expecting it to pass. For black-box testing, we've set up simple scripts, much as we did for Project 0, that run test input files you supply and compare results with output files you supply. We've set up `make check` to run both sets of tests.

We will expect your finished programs to be workmanlike—consistently indented, well-commented, readably spaced. Don't leave debugging print statements lying around. In fact, don't use them; learn to use the debugger (either `gjdb` or that of `Eclipse` or your favorite Java-system vendor).

6 Advice

First, *read and comprehend* the skeleton. Reading code is a good way to learn, so read it even if you don't use it (nothing in the skeleton is obligatory). Assume that things that puzzle you about the skeleton are there deliberately to induce you to go look things up; don't allow them to remain mysteries to you.

Speed here is not of the essence. Use simple data structures, like lists and arrays, and straightforward algorithms. This is not an exercise in cleverness, just good sense.

To the extent possible, tackle things one class and method at a time. Use unit testing to check things even when the program is incomplete. Don't go for long stretches with your program in a non-compilable state. Try to keep it correct Java as much of the time as possible, so that at any time you can run it. Likewise, run `make style` often and keep the code you add consistent with style rules. It will save you a last-minute desperate attempt to clean things up at submission time.

The example in Figure 1 is a poor test case; it isn't anywhere near pathological or thorough enough. Be sure that you do much better. Don't think that a single test will suffice.

Finally, here as in all projects, get started as soon as possible.