regexfilter.java File Reference

This application implements a text filter for multiple replacements using regular expressions (regex).

General purpose filter, replaces regular expressions (regex) in the input stream, in block or line modes.
It works like egrep and sed, or like Perl regex, but it is simpler and it can be used on Unix/Linux and Windows O.S.

Precondition:

- input: standard input (allows piping) or file.
- output: standard output or file
- regex: all pairs regex/replacement are in a file.
- works line by line or on whole input.
- macro expansions in replacement strings.

Author:
M. Sillano (marco.sillano@gmail.com)

Version:
4.02 10/11/25 (c) M.Sillano 2006-2011

Filters the input stream, via java.String.replaceAll(), using regex/replacement pairs.
All regex/replacement pairs are stored in an ASCII file, and they are all processed in sequence for every input line or for the whole input in block mode.

use

```
Usage:     regexfilter -h|-?|--help|--version",
           regexfilter [-bx] [datFile] [-i=FILE] [-u=FILE]",


           Filters the inputFile, using regex/replacement pairs in datFile.

           options:  -h|-?|--help    display this help and exit.
                     --version       print version and exit.
                     -b              block mode. Default = line mode.
                     -x              datFile in XML mode. Default = plain text.
           -i=FILE, --input=FILE     the text input file.
                                     Default = standard input.
           -o=FILE, --output=FILE    the text output file.
                                     Default = standard output.
           datFile:                  regex/replacement pairs file. Default = ./regexfilter.dat.
                                     if not found, builds an example file.
```

datFile - syntax:

- regex must be double-escaped ('\\') using regex rules (not in XML mode)
- in replacement some chars must be single-escaped (\", \', \\,\t, \f, \n, \r [*for: ", ',\, TAB, FF, LF, CR*]) and same for hexadecimal unicode: \ uxxxx. Some must be double-escaped (\\space, \\$ [*for : initial-final spaces, $ char*]).
- in replacement $n is a references to 'n' captured subsequences.
- replacement allows empty lines (delete)

- macros in replacement (processed before regexp):
  - \\space *is for intial/final spaces*
  - $V *is regexfilter version*
  - $F *is full path of the input file*
  - $P *is the path (file excluded) of the input file*
  - $N *is the file name (no ext.) of the input file*
  - $E *is the extension (no dot) of the input file*
  - $D *is the filter (file DAT) name (no ext.)*
  - $C *is the Copyright notice (CC BY-SA)*
  - $T *is the timestamp*

datFile - text format

```
# rule 1: replaces "# /**" with "/**"
 regex1=^#[ ]*(/\\*\\*)
 replacement1=$1


 # rule 302: replaces "# *"   with "*"
 regex302=^#[ ]*\\*
 replacement302=*
```

note: regex/replacement indexes MUST be in 1..999 interval.

datFile - XML format

```
<comment> rule 1: replaces "# /**" with "/**"  </comment>
 <entry key="regex1">^#[ ]*(/\*\*)</entry>
 <entry key="replacement1">$1</entry>
 <comment> rule 2: replaces "# *" with "*"  </comment>
 <entry key="regex2">^#[ ]*\*</entry>
 <entry key="replacement2">*</entry>
```

note: regex/replacement indexes MUST start from 1 and MUST be all consecutive.

Summary of regular-expression constructs

(from JDK 6 documentation)

Construct	Matches
Characters	
x	The character *x*
\\	The backslash character
\0*n*	The character with octal value 0*n* (0 <= *n* <= 7)
\0*nn*	The character with octal value 0*nn* (0 <= *n* <= 7)
\0*mnn*	The character with octal value 0*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7)
\x*hh*	The character with hexadecimal value 0x*hh*
\u*hhhh*	The character with hexadecimal value 0x*hhhh*
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')
\r	The carriage-return character ('\u000D')
\f	The form-feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')

`\c`*x*	The control character corresponding to *x*

Character classes

`[abc]`	`a`, `b`, or `c` (simple class)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	`a` through `z` or `A` through `Z`, inclusive (range)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`: `[a-dm-p]` (union)
`[a-z&&[def]]`	`d`, `e`, or `f` (intersection)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`: `[a-lq-z]`(subtraction)

Predefined character classes

`.`	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A withespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-withespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`

POSIX character classes (US-ASCII only)

`\p{Lower}`	A lower-case alphabetic character: `[a-z]`
`\p{Upper}`	An upper-case alphabetic character:`[A-Z]`
`\p{ASCII}`	All ASCII:`[\x00-\x7F]`
`\p{Alpha}`	An alphabetic character:`[\p{Lower}\p{Upper}]`
`\p{Digit}`	A decimal digit: `[0-9]`
`\p{Alnum}`	An alphanumeric character:`[\p{Alpha}\p{Digit}]`
`\p{Punct}`	Punctuation: One of `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`
`\p{Graph}`	A visible character: `[\p{Alnum}\p{Punct}]`
`\p{Print}`	A printable character: `[\p{Graph}\x20]`
`\p{Blank}`	A space or a tab: `[ \t]`
`\p{Cntrl}`	A control character: `[\x00-\x1F\x7F]`
`\p{XDigit}`	A hexadecimal digit: `[0-9a-fA-F]`
`\p{Space}`	A withespace character: `[ \t\n\x0B\f\r]`

java.lang.Character classes (simple java character type)

`\p{javaLowerCase}`	Equivalent to java.lang.Character.isLowerCase()
`\p{javaUpperCase}`	Equivalent to java.lang.Character.isUpperCase()
`\p{javawithespace}`	Equivalent to java.lang.Character.iswithespace()
`\p{javaMirrored}`	Equivalent to java.lang.Character.isMirrored()

Classes for Unicode blocks and categories

`\p{InGreek}`	A character in the Greek block (simple bloc)
`\p{Lu}`	An uppercase letter (simple category)
`\p{Sc}`	A currency symbol
`\P{InGreek}`	Any character except one in the Greek block (negation)
`[\p{L}&&[^\p{Lu}]]`	Any letter except an uppercase letter (subtraction)

Boundary matchers

`^`	The beginning of a line
`$`	The end of a line
`\b`	A word boundary

`\B`	A non-word boundary
`\A`	The beginning of the input
`\G`	The end of the previous match
`\Z`	The end of the input but for the final terminator, if any
`\z`	The end of the input

Greedy quantifiers

X`?`	*X*, once or not at all
X`*`	*X*, zero or more times
X`+`	*X*, one or more times
X`{`*n*`}`	*X*, exactly *n* times
X`{`*n*`,}`	*X*, at least *n* times
X`{`*n*`,`*m*`}`	*X*, at least *n* but not more than *m* times

Reluctant quantifiers

X`??`	*X*, once or not at all
X`*?`	*X*, zero or more times
X`+?`	*X*, one or more times
X`{`*n*`}?`	*X*, exactly *n* times
X`{`*n*`,}?`	*X*, at least *n* times
X`{`*n*`,`*m*`}?`	*X*, at least *n* but not more than *m* times

Possessive quantifiers

X`?+`	*X*, once or not at all
X`*+`	*X*, zero or more times
X`++`	*X*, one or more times
X`{`*n*`}+`	*X*, exactly *n* times
X`{`*n*`,}+`	*X*, at least *n* times
X`{`*n*`,`*m*`}+`	*X*, at least *n* but not more than *m* times

Logical operators

XY	*X* followed by *Y*
X`|`*Y*	Either *X* or *Y*
`(`*X*`)`	X, as a capturing group

Back references

`\`*n*	Whatever the n^{th} capturing group matched

Quotation

`\`	Nothing, but quotes the metacharacters: `<([{\^-=$!|]})?*+.>`
`\Q`	Nothing, but quotes all characters until `\E`
`\E`	Nothing, but ends quoting started by `\Q`

Special constructs (non-capturing)

`(?:`*X*`)`	*X*, as a non-capturing group
`(?idmsux-idmsux)`	Nothing, but turns match flags i d m s u x on - off
`(?idmsux-idmsux:`*X*`)`	*X*, as a non-capturing group with the given flags i d m s u x on - off
`(?=`*X*`)`	*X*, via zero-width positive lookahead
`(?!`*X*`)`	*X*, via zero-width negative lookahead
`(?<=`*X*`)`	*X*, via zero-width positive lookbehind
`(?<!`*X*`)`	*X*, via zero-width negative lookbehind
`(?>`*X*`)`	*X*, as an independent, non-capturing group

line terminators

The following are recognized as line terminators in regex:

- A newline (line feed) character ("\n"), - UNIX
- A carriage-return character followed immediately by a newline character ("\r\n"), - DOS/WIN
- A standalone carriage-return character ("\r"), - MAC
- A next-line character ("\u0085Q"),
- A line-separator character ("\u2028")
- A paragraph-separator character ("\u2029").

flag expressions (?idmsux-idmsux)

- In *block mode* [-b] the expressions "^" and "$" match at the beginning and the end of the entire input sequence. The embedded flag expression "(?m)" enables multiline mode, so these expressions match just after or just before, respectively, any line terminator.
- In *dotall mode*, the expression "." matches any character, including a line terminator. By default this expression does not match line terminators. Dotall mode can be enabled via the embedded flag expression (?s).
- *Case-insensitive* matching for ASCII can be enabled via the embedded (?i).
- *Unicode-aware* case folding can be enabled via the embedded flag expression (?u). Requires (?i).
- *Comments mode* can be enabled via the embedded flag expression (?x). In this mode, whitespace is ignored, and embedded comments starting with # are ignored until the end of a line.
- *Unix-lines* mode can be enabled via the embedded flag expression (?d). In this mode only "\n" line terminator is recognized in the behavior of ".","^", and "$".