

# Essential One-Liners

Walter C. Mankowski

Department of Computer Science  
Drexel University  
Philadelphia, PA

PLUG West  
July 21, 2008

# Why One-Liners?

- Perl's got a reputation for producing unreadable, unmaintainable code.

# Why One-Liners?

- Perl's got a reputation for producing unreadable, unmaintainable code.
- A lot of work has been done on tools and techniques to get around that.

# Why One-Liners?

- Perl's got a reputation for producing unreadable, unmaintainable code.
- A lot of work has been done on tools and techniques to get around that.
- Perl's still great for throwing together quick and dirty little programs.

# Why One-Liners?

- Perl's got a reputation for producing unreadable, unmaintainable code.
- A lot of work has been done on tools and techniques to get around that.
- Perl's still great for throwing together quick and dirty little programs.
- Nothing's quicker and dirtier than the one-liner.

# But I'm a Java programmer!

# But I'm a Java programmer!

- Text files are *everywhere*.

## But I'm a Java programmer!

- Text files are *everywhere*.
- The Unix command-line environment is incredibly powerful:

## But I'm a Java programmer!

- Text files are *everywhere*.
- The Unix command-line environment is incredibly powerful:
  - everything is ASCII

## But I'm a Java programmer!

- Text files are *everywhere*.
- The Unix command-line environment is incredibly powerful:
  - everything is ASCII
  - create new “programs” by connecting small, simple existing programs in pipelines

## But I'm a Java programmer!

- Text files are *everywhere*.
- The Unix command-line environment is incredibly powerful:
  - everything is ASCII
  - create new “programs” by connecting small, simple existing programs in pipelines
  - less, wc, sort, xargs, sed, tr, cut, tee, etc.

## But I'm a Java programmer!

- Text files are *everywhere*.
- The Unix command-line environment is incredibly powerful:
  - everything is ASCII
  - create new “programs” by connecting small, simple existing programs in pipelines
  - less, wc, sort, xargs, sed, tr, cut, tee, etc.
- Perl fits really well into this niche.

# Freeing your time for more important things . . .

# Freeing your time for more important things . . .



# Freeing your time for more important things . . .



# Perl One-Liners 101

# Perl One-Liners 101

(Cell phones off?)

# Hello, world

```
% perl -e 'print "Hello, world.\n"'
```

# Hello, world

```
% perl -e 'print "Hello, world.\n"'  
Hello, world.  
%
```

# Hello, world

```
% perl -e 'print "Hello, world.\n"'  
Hello, world.  
%
```

- `-e` to enter the program on the command line

# Hello, world

```
% perl -e 'print "Hello, world.\n"'  
Hello, world.  
%
```

- -e to enter the program on the command line
- enclose program in **single quotes** to avoid shell expansion

# Hello, world

```
% perl -e 'print "Hello, world.\n"'  
Hello, world.  
%
```

- -e to enter the program on the command line
- enclose program in single quotes to avoid shell expansion
- you don't need the final semicolon

# Hello, world

```
% perl -e 'print "Hello, world.\n"'  
Hello, world.  
%
```

- -e to enter the program on the command line
- enclose program in single quotes to avoid shell expansion
- you don't need the final semicolon
- no strict

# Hello, world

```
% perl -e 'print "Hello, world.\n"'  
Hello, world.  
%
```

- -e to enter the program on the command line
- enclose program in single quotes to avoid shell expansion
- you don't need the final semicolon
- no strict
- no warnings

# Hello, world

```
% perl -e 'print "Hello, world.\n"'  
Hello, world.  
%
```

- -e to enter the program on the command line
- enclose program in single quotes to avoid shell expansion
- you don't need the final semicolon
- no strict
- no warnings
- no tests

# That's it!

- That's all you need to know.

# That's it!

- That's all you need to know.
- The rest of this talk is all about syntactic sugar to make one-liners easier to write.

# Perl programmers **love** syntactic sugar

# Perl programmers **love** syntactic sugar



# Automatic newlines with `-l`

The `-l` flag automatically adds a newline to whatever you print.

Without `-l`

```
perl -e 'print "Hello, world.\n"'
```

# Automatic newlines with `-l`

The `-l` flag automatically adds a newline to whatever you print.

Without `-l`

```
perl -e 'print "Hello, world.\n"'
```

With `-l`

```
perl -le 'print "Hello, world."'
```

# say what?

Perl 5.10 introduced a new builtin function, **say**, that works just like `print` except that it automatically adds a newline.

# say what?

Perl 5.10 introduced a new builtin function, **say**, that works just like `print` except that it automatically adds a newline.

Sadly, `say` doesn't work with `-e`:

```
% perl -e 'say "Hello, world."'
```

```
String found where operator expected at -e line 1, near  
"say "Hello, world.""
```

(Do you need to predeclare `say`?)

```
syntax error at -e line 1, near "say "Hello, world.""
```

```
Execution of -e aborted due to compilation errors.
```

```
%
```

## say what? (continued)

To avoid breaking backward compatibility, `say` is turned off by default in 5.10.

## say what? (continued)

To avoid breaking backward compatibility, `say` is turned off by default in 5.10.

To turn it on from the command line, use `-E` instead of `-e`:

```
% perl -E 'say "Hello, world."'
Hello, world.
%
```

# Writing Loops

Suppose you want to see if your team is following your new coding standards that lines can't be longer than 80 characters. Here's one way to write that:

```
perl -e 'while (<>) {print if length > 80}' *.pl
```

# The `-n` flag

That gets tedious to write that all the time, so perl has a `-n` flag that automatically puts a loop around your code. It's equivalent to

```
while {<>} {  
    ... # your code goes here  
}
```

## The `-n` flag

That gets tedious to write that all the time, so perl has a `-n` flag that automatically puts a loop around your code. It's equivalent to

```
while {<>} {  
    ... # your code goes here  
}
```

### Without `-n`

```
perl -e 'while (<>) {print if length > 80}' *.pl
```

## The `-n` flag

That gets tedious to write that all the time, so perl has a `-n` flag that automatically puts a loop around your code. It's equivalent to

```
while {<>} {  
    ... # your code goes here  
}
```

### Without `-n`

```
perl -e 'while (<>) {print if length > 80}' *.pl
```

### With `-n`

```
perl -ne 'print if length > 80' *.pl
```

## BEGIN and END blocks

If you want to do pre- or post-processing when using the `-n` flag, use BEGIN and END blocks.

For example, if the file `nums` contains

```
1
2
3
4
```

## BEGIN and END blocks (continued)

### Sum

```
% perl -lne '$s += $_; END{print $s}' nums  
10  
%
```

# BEGIN and END blocks (continued)

## Sum

```
% perl -lne '$s += $_; END{print $s}' nums
10
%
```

## Product

```
% perl -lne 'BEGIN{$p = 1} $p *= $_; END{print $p}' \
  nums
24
%
```

## Writing Loops (continued)

Suppose you want to convert an existing file to lowercase. Now that you know about the `-n` flag, you might try writing it like this:

```
perl -ne 'tr/A-Z/a-z/; print' foo
```

## Writing Loops (continued)

Suppose you want to convert an existing file to lowercase. Now that you know about the `-n` flag, you might try writing it like this:

```
perl -ne 'tr/A-Z/a-z/; print' foo
```

```
perl -ne 'tr/A-Z/a-z/; print' foo >foo.out
```

# The `-p` flag

Printing each line is a common enough operation that perl has a special flag for it, the `-p` flag. It's equivalent to

```
while {<>} {  
    ... # your code goes here  
} continue {  
    print or die "-p destination: $!\n";  
}
```

## The `-p` flag

Printing each line is a common enough operation that perl has a special flag for it, the `-p` flag. It's equivalent to

```
while {<>} {  
    ... # your code goes here  
} continue {  
    print or die "-p destination: $!\n";  
}
```

It's similar to the `-n` flag, except `-p` prints out each line:

### Without `-p`

```
perl -ne 'tr/A-Z/a-z/; print' foo
```

## The `-p` flag

Printing each line is a common enough operation that perl has a special flag for it, the `-p` flag. It's equivalent to

```
while {<>} {  
    ... # your code goes here  
} continue {  
    print or die "-p destination: $!\n";  
}
```

It's similar to the `-n` flag, except `-p` prints out each line:

### Without `-p`

```
perl -ne 'tr/A-Z/a-z/; print' foo
```

### With `-p`

```
perl -pe 'tr/A-Z/a-z/' foo
```

# Advanced Perl One-Liners

## Editing files in-place

Instead of just printing out the file, one thing you might want to do with the `-p` flag is make the changes directly to the file. Perl's `-i` flag does exactly that:

## Editing files in-place

Instead of just printing out the file, one thing you might want to do with the `-p` flag is make the changes directly to the file. Perl's `-i` flag does exactly that:

Print to stdout

```
perl -pe 's/foo/bar/' a.pl
```

## Editing files in-place

Instead of just printing out the file, one thing you might want to do with the `-p` flag is make the changes directly to the file. Perl's `-i` flag does exactly that:

### Print to stdout

```
perl -pe 's/foo/bar/' a.pl
```

### Edit a.pl in-place

```
perl -pi -e 's/foo/bar/' a.pl
```

## Editing files in-place

Instead of just printing out the file, one thing you might want to do with the `-p` flag is make the changes directly to the file. Perl's `-i` flag does exactly that:

### Print to stdout

```
perl -pe 's/foo/bar/' a.pl
```

### Edit a.pl in-place

```
perl -pi -e 's/foo/bar/' a.pl
```

### Editing multiple files

```
perl -pi -e 's/foo/bar/' *.pl
```

## Editing files in-place

Obviously the `-i` flag is dangerous since it clobbers whatever was originally in the file. So Perl lets you specify a backup file when using `-i`.

## Editing files in-place

Obviously the `-i` flag is dangerous since it clobbers whatever was originally in the file. So Perl lets you specify a backup file when using `-i`.

Edit `a.pl` in-place

```
perl -pi -e 's/foo/bar/' a.pl
```

## Editing files in-place

Obviously the `-i` flag is dangerous since it clobbers whatever was originally in the file. So Perl lets you specify a backup file when using `-i`.

### Edit a.pl in-place

```
perl -pi -e 's/foo/bar/' a.pl
```

### Original file saved in a.pl.bak

```
perl -p -i.bak -e 's/foo/bar/' a.pl
```

## Automatically splitting files

Use the **-a flag** to automatically split each line (like AWK).  
Default is to split on ' '; use the **-F flag** to split on something else.

## Automatically splitting files

Use the **-a flag** to automatically split each line (like AWK).  
Default is to split on ' '; use the **-F flag** to split on something else.

Print processes whose parents are init

```
ps axl | perl -ane 'print if $F[3] == 1'
```

## Automatically splitting files

Use the **-a flag** to automatically split each line (like AWK).  
Default is to split on ' '; use the **-F flag** to split on something else.

Print processes whose parents are init

```
ps axl | perl -ane 'print if $F[3] == 1'
```

Print all userids and user names

```
perl -aln -F: -e 'print "$F[2]\t$F[0]"' /etc/passwd
```

## Using modules

Instead of explicitly using a module, you can load a module from the command line with the **-M flag**.

The following programs both do the same thing:

### Use module

```
perl -e 'use LWP::Simple; getprint "http://pghpw.org"'
```

# Using modules

Instead of explicitly use'ing a module, you can load a module from the command line with the **-M flag**.

The following programs both do the same thing:

## Use module

```
perl -e 'use LWP::Simple; getprint "http://pghpw.org"'
```

## -M flag

```
perl -MLWP::Simple -e 'getprint "http://pghpw.org"'
```

## Input record separators

Perl normally reads input until it hits the “input record separator”, which defaults to `\n` and can be changed by setting `$/`. But in addition to setting `$/` in a `BEGIN` block, you can also change it on the command line with the **`-0` flag**. It sets `$/` to an octal or hex number:

## Input record separators

Perl normally reads input until it hits the “input record separator”, which defaults to `\n` and can be changed by setting `$/`.

But in addition to setting `$/` in a BEGIN block, you can also change it on the command line with the **-0 flag**. It sets `$/` to an octal or hex number:

`-0x0d` carriage returns

## Input record separators

Perl normally reads input until it hits the “input record separator”, which defaults to `\n` and can be changed by setting `$/`.

But in addition to setting `$/` in a `BEGIN` block, you can also change it on the command line with the **`-0` flag**. It sets `$/` to an octal or hex number:

```
-0x0d  carriage returns  
-0      null character (find -print0)
```

# Input record separators

Perl normally reads input until it hits the “input record separator”, which defaults to `\n` and can be changed by setting `$/`.

But in addition to setting `$/` in a `BEGIN` block, you can also change it on the command line with the **-0 flag**. It sets `$/` to an octal or hex number:

- `-0x0d` carriage returns
- `-0` null character (`find -print0`)
- `-00` paragraph mode (useful for Postfix logs)

## Input record separators

Perl normally reads input until it hits the “input record separator”, which defaults to `\n` and can be changed by setting `$/`.

But in addition to setting `$/` in a `BEGIN` block, you can also change it on the command line with the **-0 flag**. It sets `$/` to an octal or hex number:

- `-0x0d` carriage returns
- `-0` null character (`find -print0`)
- `-00` paragraph mode (useful for Postfix logs)
- `-0777` slurp in entire file

# Summary of flags

Flag	Result
-e	Execute program on command line
-l	Automatically add newlines
-n	Automatically loop
-p	Automatically loop and print each line
-i	Edit files in-place
-a	Automatically split input
-M	Use module
-0	Change input record separator

## More Information

- `perldoc perlrun`
  - many more features than I covered

## More Information

- `perldoc perlrun`
  - many more features than I covered
- Google for “perl one liners”
  - Tom Christiansen one-liners
  - article by Jeff Bay in *The Perl Review*
  - “One-liners 101” on IBM developerWorks
  - many others

# Java doesn't have one-liners

## DOCTOR FUN

21 Apr 2005



Copyright © 2005 David Farley, d-farley@ibiblio.org  
<http://ibiblio.org/Dave/drfun.html>

This cartoon is made available on the Internet for personal viewing only. Opinions expressed herein are solely those of the author.

Quentin Tarantino's "Learn Java in a Minute"

# Examples!

# Palindromes

## Find palindromes

```
perl -lne 'print if $_ eq reverse' \  
/usr/share/dict/words
```

## Line Numbers

Print lines preceded by line number

```
perl -ne 'print "$. $_"'
```

`$.` is a special Perl variable that contains the input line number.

## Computing Averages

Sum lines, then divide by total number of lines

```
% perl -lne '$s += $_; END{print $s/$.}' nums  
2.5  
%
```

## Printing selected lines

Print lines 10–20

```
perl -ne 'print if 10..20'
```

The `..` operator is magic when used in scalar context. Read the “Range Operators” section in `perlop`.

## Grep with Perl regular expressions

Poor man's grep

```
perl -ne 'print if /^foobar/'
```

## Grep with Perl regular expressions

### Poor man's grep

```
perl -ne 'print if /^foobar/'
```

...but with the full power of Perl's regular expressions!

### Lines with foo not followed by bar

```
perl -ne 'print if /foo(?!bar)/'
```

## Grep with Perl regular expressions

### Poor man's grep

```
perl -ne 'print if /^foobar/'
```

...but with the full power of Perl's regular expressions!

### Lines with foo not followed by bar

```
perl -ne 'print if /foo(?!bar)/'
```

### Lines with bar not preceded by foo

```
perl -ne 'print if /(?!foo)bar/'
```

## Grep with Perl regular expressions

### Poor man's grep

```
perl -ne 'print if /^foobar/'
```

...but with the full power of Perl's regular expressions!

### Lines with foo not followed by bar

```
perl -ne 'print if /foo(?!bar)/'
```

### Lines with bar not preceded by foo

```
perl -ne 'print if /(?!foo)bar/'
```

### Grep on paragraphs

```
perl -00 -ne 'print if /foo(?!bar)/'
```

## Random Numbers

Does `rand()` return the same sequence with the same seed on different platforms?

## Random Numbers

Does `rand()` return the same sequence with the same seed on different platforms?

Look at the first 5

```
% perl -le 'srand(42); print rand for 1..5'  
0.744525000061007  
0.342701478718908  
0.111085282444161  
0.422338957988309  
0.0811111711783106
```

## Random Numbers

Does `rand()` return the same sequence with the same seed on different platforms?

Look at the first 5

```
% perl -le 'srand(42); print rand for 1..5'  
0.744525000061007  
0.342701478718908  
0.111085282444161  
0.422338957988309  
0.0811111711783106
```

Try a whole bunch

```
% perl -le 'srand(42); print rand for 1..100_000' \  
| md5sum  
ac18d07f40c858bf4b23090177f6a685  -
```

## Stacking the deck

One-liners can also be used in shell scripts

```
#!/bin/bash

SEED='perl -le 'print int rand 0xffffffff''

for ((n = 10; n <= 400; n += 10)) do
    cmd="./wn_path_seq $n $SEED"
    echo $cmd
    $cmd
done
```

## Add lines to file

Add line to beginning of file

```
perl -0777 -i -ne 'print "first\n$_"' test.txt
```

## Add lines to file

Add line to beginning of file

```
perl -0777 -i -ne 'print "first\n$_"' test.txt
```

Same thing

```
perl -0777 -i -pe '$_ = "first\n$_"' test.txt
```

## Add lines to file

Add line to beginning of file

```
perl -0777 -i -ne 'print "first\n$_"' test.txt
```

Same thing

```
perl -0777 -i -pe '$_ = "first\n$_"' test.txt
```

Same thing, more obfuscated

```
perl -0777 -i -pe 's//first\n/' test.txt
```

# Thank you!