EASY AS SIX

by Arne Sommer

PerlCon 2019 - Riga

https://perl6.eu/easy-as-six.pdf

Corrections and comments: https://perl6.eu/easy-as-six.html

EASY AS SIX

Why write a lot of code when you can let Perl 6 do most of the job for you?

Programmers repeat themselves all the time. Reinventing the wheel can be fun, but modules and object orientation are helpful in reducing the code base.

EASY AS SIX/2

The Perl 6 designers took a hard look at the common tasks, and came up with ways to reduce the amount of code we have to write. The result is a lot of very powerful keywords in the core language.

This talk takes a look at some of them.

WHAT IT ISN'T ABOUT

THIS IS NOT AN INTRODUCTION TO PERL 6

See my «Perl6 In 45+45 Minutes» introduction held at the Nordic Perl Worskshop in Oslo in September 2018.

https://npw2018.perl6.eu/

(PDF + source code)

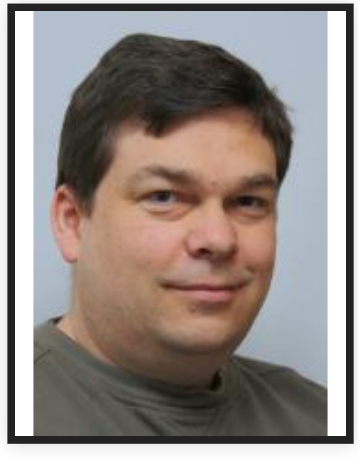
ABOUT ME

ARNE SOMMER

I have programmed Perl for 30 years...

Perl 3 and 4 (1989-1995), Perl 5 (1994-), and Perl 6 (2015-).

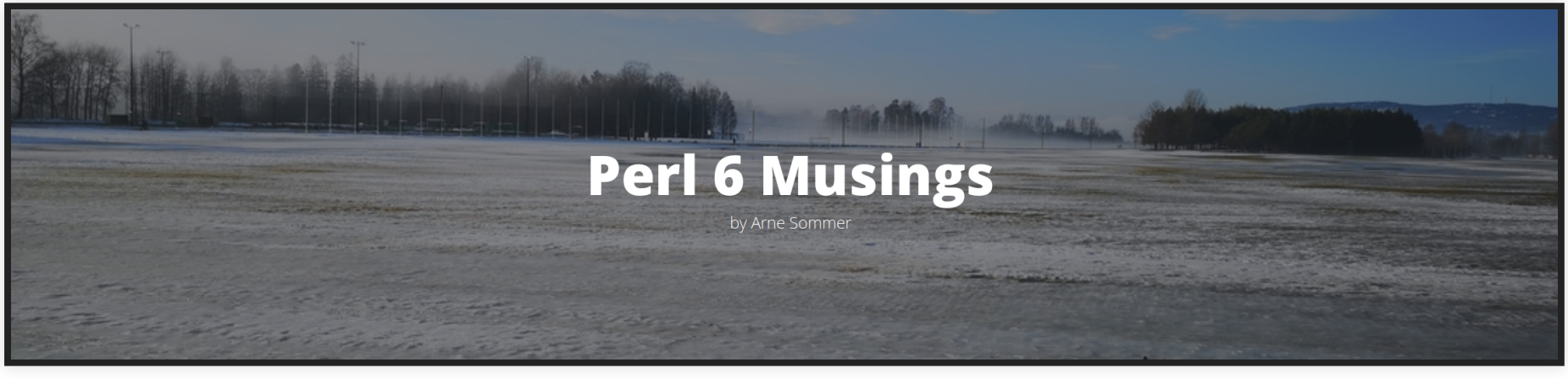
I am active in Oslo.pm (in Norway).



Web: bbop.org
Email: arne@bbop.org
CPAN: ARNE
GitHub: arnesom

PERL 6 BLOG

I have a Perl 6 Blog, at https://perl6.eu



Article #25 published today!

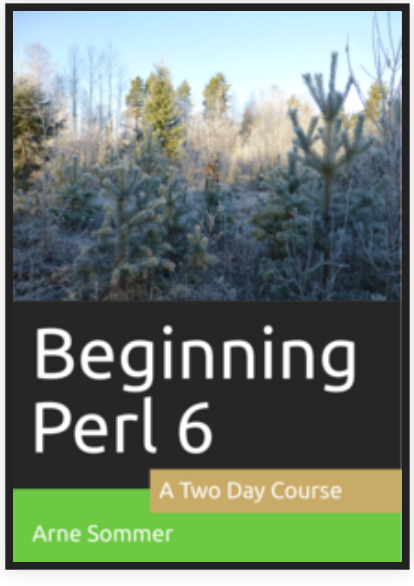
PERL WEEKLY CHALLENGE

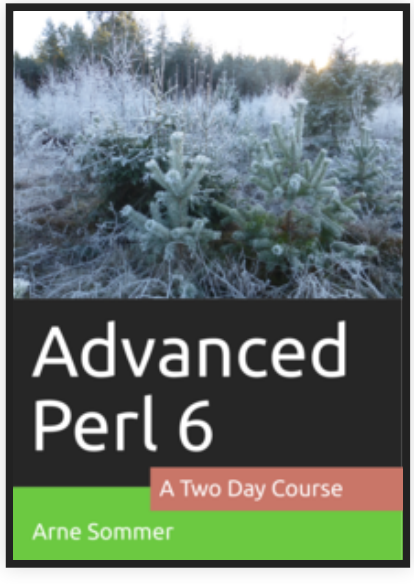
Most of the articles are in response to the «Perl Weekly Challenge».

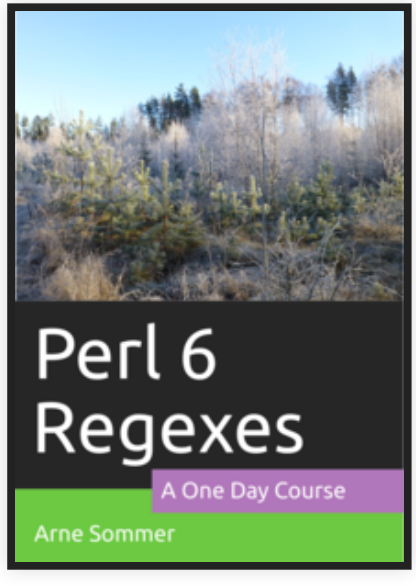
https://perlweeklychallenge.org

PERL 6 BOOK & COURSES

I am working on several Perl 6 courses:







https://course.perl6.eu

BEGINNING PERL 6

The «Beginning Perl 6» course in Riga was cancelled due to lack of interest.

NEWLINES

OUTPUT

Perl 5:

```
print "Hello!\n";
```

Perl 6:

```
say "Hello!";
```

«say» gives you a trailing newline automatically.

(«print» is available as well, if you need manual control of the newlines.)

INPUT

Read a line from the keyboard, without the trailing newline.

Perl 5:

```
chomp(my $a = get);
```

Perl 6:

```
my $a = get;
```

IN- AND OUTPUT

Perl 5:

```
print "Name: ";
chomp(my $name = get);
```

Perl 6:

```
my $name = prompt "Name: ";
```

FILES

NEWLINE != NEWLINE

A newline is one or two characters:

OS	Character(s)	Strings	Codepoint
Windows	<CR><LF>	\r\n	10 + 13
Linux	<LF>	\n	10
Mac OSX	<LF>	\n	10
Mac (old)	<CR>	\r	13

Perl 6 takes care of them all.

READING A FILE

Perl 5:

```
if (open($fh, "filename"))
{
  while (my $line = <$fh>);
  {
    chomp $line; do_something($line);
  }
  close $fh;
}
```

Perl 6:

```
do-something($line) for "filename".IO.lines;
```

«IO.lines» opens and closes the file automatically.

READING A FILE/2

If you want the entire file:

Perl6:

```
$content = slurp "filename";
```

One single string, with all the newlines intact.

We can get the individual lines like this:

```
do-something($_) for $content.lines;
```

WRITING A FILE

Perl 5:

```
my @lines = ...;
if (open(my $fh, '>', "filename"))
{
  print $fh $_\n" for @lines;
  close $fh;
}
```

Perl 6:
The same way. But...

WRITING A FILE/2

If you want to write the entire file at once:

Perl 6:

```
spurt "filename", $text;
```

DOES THE FILE ON THE COMMAND LINE EXIST?

Perl 5:

```
my $file = $ARGV[0] || die("Specify a file");
if (-e $file && -r $file)
{
  do_something;
}
```

Perl 6:

```
sub MAIN ($file where $file.IO.e && $file.IO.r)
{
  do-something;
}
```

IF (AND ALTERNATIVES)

STACKED CONDITIONS

Perl 5:

```
do_something($a) if ($a >= 5 && $a <= 10);
```

Don't write this:

```
do_something($a) if ($a >= 5 and $a <= 10);
```

Perl 6:

```
do-something($a) if 5 <= $a <= 10;
```

AVOIDING IF

Perl 5 (with «if»):

```
sub do_something
{
  my $a = shift;
  if ($a == 42) { print "The meaning of ..\n" }
  else          { print "$a\n"; }
}
```

Perl 6 (With Multiple Dispatch):

```
multi sub do_something (42) { say "The meaning of .."; }
multi sub do-something ($a) { say $a; }
```

The first one is a short form for:

```
multi sub do_something ($a where $a == 42)
```

JUNCTIONS

COMPLICATED IFS

Perl 5:

```
do_something($a) if ($a == 1 || $a == 6 || $a == 11);
```

Perl 6:

With Junctions

```
do-something($a) if $a == 1 | 6 | 11;
```

Easier to read, harder to make an error (typing $b) and scales quite well.

JUNCTIONS

Or we could write it like this:

```
do-something($a) if $a == any(1, 6, 11);
```

Junctions work in assignments as well:

```
my $a = 1 | 2;
say $a;                 # -> any(1, 2)

say "L" if $a == 1;   # -> L
say "M" if $a == 2;   # -> M
```

The variable «$a» has two values *at the same time*.

(Much like Schrödinger's Cat.)

THE TYPE SYSTEM

TYPES AND CLASSES

Perl 5:

No type system, except classes (with a strange syntax).

Perl 6:

- A full type system, that you can choose to use
- Everything is an object, and can be treated as such. If you want to...
- Classes (on stereoids), with a straightforward syntax

TYPES

Perl 6:

```
my Int $a = 12;

$a = "13";
Type check failed in assignment to $a;
  expected Int but got Str ("13") in block ...
```

```
my Str $b = "abc";

$b = 10;
Type check failed in assignment to $b;
  expected Str but got Int (10) in block ...
```

NUMBERS WITH DECIMAL POINTS

You may have seen this yesterday:

```
perl -E 'say 0.1 + 0.2 - 0.3'
5.55111512312578e-17
```

```
perl6 -e 'say 0.1 + 0.2 - 0.3'
0
```

Cobol got it right, but then it went downhill. Until Perl 6...

RATIONAL NUMBERS

```
my $a = 1/3;
my $b = $a + $a + $a;
say $a;
```

Most languages:

```
0.99999999999999999 # Or similar
```

Perl 6:

```
1
```

Disclamer: Some languages may have followed Perl 6's lead. (Perl 5 has...)

RATIONAL NUMBERS/2

```
my $a = 1/3;
say $a;        # -> 0.333333
say $a.WHAT;   # -> (Rat)
say $a.perl;   # -> <1/3>
```

A «Rat» value is stored as two integers; the numerator («$a.numerator») and the denomerator («$a.denomerator»), and the value that the first «say» gives us is evaluated just for printing it (stringification).

Use «$a.nude» to get both.

And don't say that the Perl 6 developers lack humor.

RATIONAL NUMBERS/3

Note that a «Rat» reduces the values as much as possible:

```
my $a = 3/9;
say $a.perl;  # -> <1/3>

my $b = 1/5;
say $b.perl;  # -> <1/5>

my $c = $a + $b
say $c;       # -> 0.533333
say $c.perl;  # -> ???
```

The last one gives?

```
say $c.perl;  # -> <8/15>
```

INTEGERS (INT)

Using the type system gives error checking for free:

Perl 6:

```
sub addition (Int $a, Int $b) { return $a + $b; }
my $sum = addition("one", "ten");
```

Result:

```
===SORRY!=== Error while compiling:
Calling addition(Str, Str) will never work with declared \
  signature (Int $a, Int $b)
------> <BOL>⏏addition("one", "ten");
```

The error is catched at *Compile Time*.

WITHOUT TYPES

Without the types, we get a Run Time error:

Perl 6:

```
sub addition ($a, $b) { return $a + $b; }
my $sum = addition("one", "ten");
```

Result:

```
Cannot convert string to number: base-10 number must begin \
with valid digits or '.' in '⏏one' (indicated by ⏏)
  ...
```

REPEATING TYPE CONSTRAINTS

You have a procedure that only works with prime numbers:

```
sub magic-function (Int $a where $a.is-prime)
{
  ...;
}
```

But what if you have a lot of procedures with the same requirement?

CUSTOM TYPES

Use a custom type:

```
subset Prime of Int where *.is-prime;

sub magic-function (Prime $a)
{
  say "OK";
}
```

We can skip the «Int» part:

```
subset Prime where *.is-prime;
```

CUSTOM TYPES/2

Testing it:

```
> magic-function(1);
Constraint type check failed in binding to parameter '$a'; \
    expected Prime but got Int (1)
  in sub magic-function at <unknown file> line 1
  in block <unit> at <unknown file> line 1

> magic-function(3);
OK
```

CUSTOM TYPES AND VARIABLES

Custom Types work with variables:

```
my Prime $r;
```

As do Type Constraints:

```
my Int $s where .is-prime;
my     $t where .is-prime;
```

RANDOMNESS & RANGES

A RANDOM INTEGER

A random integer from 10 to 99, both included.

Perl 6 (with «rand»):

```
my $value = rand(90).Int + 10; ## 90 == 99 - 10 + 1
```

It is very easy to get it wrong.

A RANDOM INTEGER/2

Perl6 (with «pick»):

```
my $value = (10 .. 99).pick;
```

Impossible to get it wrong.

RANGES

The «10 .. 99» construct is a *Range*. It gives consecutive integers from the first value to the last value, both included.

RANGES/2

It is possible to skip the first, last, or both values in the result by appending a «^» (caret) on the relevant side.

These are all equal:

```
my $value = (10 .. 99).pick;
my $value = (9 ^.. 99).pick;
my $value = (9 ^..^ 100).pick;
```

Note that the caret is part of the Range operator «..», without any whitespace!

RANGES SHORTCUT

A loop to execute ten times:

```
do-something($_) for 1 .. 10;
```

We can use the «but not including» caret as a shorthand:

```
do-something($_) for ^10;
```

This is also executed ten times, but...

The values are off by -1; 0 .. 9 (instead of 1 .. 10). Index-friendly...

SEQUENCES

RANGES VS SEQUENCES

A Range:

```
my $values := 1 .. Inf;
```

A Sequence:

```
my $values := 1 ... Inf;
```

A range can only hold consecutive (increasing) integers, but a sequence can hold all types of numeric values (in any order) - and strings.

Ranges & Sequences are usually lazy; the values are only calculated when actually needed.

SEQUENCES

```
my $values := 1 ... Inf;
```

Binding (:=) instead of assignment (=) is required, and you cannot bind to an array (@values).

(As the assignment to an array would expand it to a list, and it is hard to do that with infinity...)

```
say $values[3];  # -> 2
```

COMMON SEQUENCES

Perl 6 recognizes some simple sequences:

```
my $v := 1,3,5 ... Inf;
say $v[^10];  # -> (1 3 5 7 9 11 13 15 17 19)
```

```
my $v := 1,2,4 ... Inf;
say $v[^10];  # -> (1 2 4 8 16 32 64 128 256 512)
```

```
my $v := 1,3,6 ... Inf;
say $v[^10];
Unable to deduce arithmetic or geometric sequence from \
    1,3,6 (or did you really mean '..'?)
  in block <unit> at <unknown file> line 1
```

This is an example of a «deferred error». The error (actually an exception) is only triggered when you use the value.

DEFERRED ERRORS

Try the following:

```
my $a = 1 / 0;
```

It doesn't blow up, as we do not use (access) the value.

But this (the «say» line) crashes the program:

```
my $a = 1 / 0;
say $a;
```

Note that in REPL mode, the first one will cause an Exception, as the value is printed.

A PRIME SEQUENCE

We can set up a sequence of the prime numbers like this:

```
my $primes := (1 ... Inf).grep(*.is-prime);
```

The 10 first primes:

```
> say $primes[^10];
(2 3 5 7 11 13 17 19 23 29)
```

A PRIME SEQUENCE & TYPE

```
subset Prime of Int where *.is-prime;

my $primes := (^Inf).grep(*.is-prime);

my Prime $a = $primes[8];  # 23

say $a++;
Type check failed in assignment to $a; expected Prime \
    but got Int (24)
  in block <unit> at <unknown file> line 1
```

CUSTOM OPERATOR

We can set up a custom version of «++» to work on primes:

```
subset Prime of Int where *.is-prime;
multi sub postfix:<++>(Prime $n is rw)
{
  for $n ^.. *   # Start with $n+1
  {
    ($n = $_; last) if .is-prime;
  }
}
```

Using it:

```
my Prime $a = 3;

( $a++; print "$a " ) for ^10; print "\n";
# -> 5 7 11 13 17 19 23 29 31 37
```

DEFINED OR NOT

BOOLEAN VALUES

Perl 6 has the Boolean values «True» and «False» built in.

The line between Boolean values and numbers is blured:

```
say True  + 0;  # -> 1
say False + 0;  # -> 0
```

BOOLEAN CONTEXT

The Perl 5 behaviour of using any value in Boolean context is here:

Perl 6:

```
my $a = 5;   do-something if $a;          # Called
my $b = "Q"; do-something if $b;          # Called
my $c = 0;   do-something if $c;          # Not called
my $c = 0;   do-something if $c.defined; # Called
my $d;       do-something if $d;          # Not called
```

REGEXES (AND ALTERNATIVES)

REGEX 001

Perl 6 Regexes respect Unicode properties:

```
say Ⅵ ~~ /\d/;  # -> 「6」
```

A roman numeral as a single Unicode character.

```
say "ß" ~~ /<:letter>/;  # -> 「ß」
```

<:letter> is a character class. There are many of them, and they have decent names.

The funny quotes in the output are there to show us that this is a match object, and not a string.

COMPLEX REGEXES

A Perl6 Regex that parses a URL:

```
$url ~~
/^
  (<[a..z]><[a..z 0..9 + . : \-]>*)\: # $0 scheme
  [\/\/                              # //
    [(.*[\:.+]?)\@]?                 # $1 userinfo (opt.)
    (<[\w \. \-]>*)                  # $2 host
    [\:(\d+)]?                       # $3 port (optional)
    (\/?)                            # $4 path separator
  ]?                                 # $1-$4 are optional
  ([<[\w \d -] - [#?]>]+)?           # $5 path (optional)
  [\?(<[\w \d \- =]>*)]?             # $6 query (optional)
  [\#(.*)]?                          # $7 fragment (opt.)
$/
```

Sort of. It has some errors. The comments help, but it isn't very readable.

GRAMMARS

```
my $result = URL.parse($url);

grammar URL
{
  regex TOP        { <SchemeW> <Hostinfo>? <Path>?
                     <QueryW>? <FragmentW>? }
  regex SchemeW    { <Scheme> <SchemeS> }
  regex SchemeS    { ':' }
  regex Scheme     { <[a..z]><[a..z 0..9 + . : \-]>* }
  regex Hostinfo   { '//' <UserinfoW>? <Host> <PortW>? }
  regex UserinfoW  { <Userinfo> <UserinfoS> }
  regex Userinfo   { .*[\:.+]? }
  regex UserinfoS  { '@' }
  regex Host       { <[\w \. \-]>* }
```

W=Wrapper Regex
S=Separator (to get rid of).

GRAMMARS/2

The rest of it:

```
  regex PortW     { <PortS> <Port> }
  regex PortS     { ':' }
  regex Port      { \d+ }
  regex Path      { '/'? <[\w \d -] - [#?]>+ }
  regex QueryW    { <QueryS> <Query> }
  regex QueryS    { '?'  }
  regex Query     { <[\w \d \- =]>* }
  regex FragmentW { <FragmentS> <Fragment> }
  regex FragmentS { '#' }
  regex Fragment  { .+ }
}
```

See https://perl6.eu/ackerman-url.html for details.

THE PERL WEEKLY CHALLENGE #20.1

«Write a script to accept a string from command line and split it on change of character. For example, if the string is **“ABBCDEEF”**, then it should split like **“A”**, **“BB”**, **“C”**, **“D”**, **“EE”**, **“F”**.

LOOP

```
sub split-change ($in)
{
  my @out;
  my @in = $in.comb;
  my $out;

  while @in
  {
    $out = @in.shift;

    while @in
    {
      if @in[0] eq
        $out.substr(0,1)
      {
        $out ~= @in.shift;
      }
      else
      {
        @out.push($out);
        $out = "";
        last;
      }
    }
  }
  @out.push($out) if $out;
  return @out;
}
```

GATHER/TAKE

```
sub split-change ($in) {
  gather {
    my $out = $in.substr(0,1);
    for 1 .. $in.chars -> $index {
      if $out.substr(0,1) eq $in.substr($index,1) {
        $out ~= $in.substr($index,1);
      }
      else {
        take $out;
        $out = $in.substr($index,1);
      }
    }
  }
}
```

GRAMMAR

```
grammar SPLIT
{
  regex TOP  { <Char>+ }
  regex Char { (.) $0* }
}

sub split-change ($in)
{
  my $result = SPLIT.parse($in);
  return $result<Char>.map({ $_.Str });
}
```

See https://perl6.eu/amicable-split.html for details.

FINAL WORDS

LINKS

This presentation: https://perl6.eu/easy-as-six.pdf

My «Perl6 In 45+45 Minutes» introduction:
https://npw2018.perl6.eu/

My Perl 6 blog: https://perl6.eu

Any Questions?

```
exit;
```