PERL6 IN 45+45 MINUTES

☜ PART 1 ☞

by Arne Sommer

Nordic Perl Workshop
September 2018
Oslo

This presentation is available online:
https://npw2018.perl6.eu/

ABOUT ME

I have a master's degree in Computer Science.

I have programmed perl since 1989.

perl3 and 4 (1989-1995), perl5 (1994-), and perl6 (2015-).

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

ABOUT PERL6 (PART 1)

VARIABLES & VALUES (PART 1)

PROCEDURES (PART 2)

FILES & IO (PART 2)

RAKUDO PERL6

Rakudo is a production ready implementation of **Perl6**, written in **NQP** («Not Quite Perl»), running on the dedicated **MoarVM** («Metamodel On A Runtime») virtual machine.

Rakudo has monthly releases.

Implementations of Rakudo running on jvm and javascript (node.js) are not as complete.

RAKUDO STAR

Rakudo Star is released every third month. This is Rakudo bundled with documentation (the «p6doc» command) and a selection of useful modules (especially the module installer «zef»).

VOLUNTEERS

The perl6 specification and development process has been done by volunteers. There are no company or rich uncle behind this.

That is quite impressive!

SPEED

Rakudo is generally slower than perl5, but much faster than just a year ago.

The developer focus has been: «Make it right, then make it fast».

Perl6 is fully Unicode compliant, making it slower than it could have been.

MORE INFORMATION

- «p6doc» online: https://docs.perl6.org
- The excellent «Perl6 Weekly» gives weekly summaries of (almost) everything related to perl6: https://p6weekly.wordpress.com/
- Ask questions on perl6 issues on the #perl6 IRC channel on irc.freenode.net. See https://perl6.org/community/irc for more information

REPL

Run perl6 without any arguments to start it in interactive, or REPL (Read-Eval-Print Loop), mode.

```
$ perl6
To exit type 'exit' or '^D'
> say 12; my $a = False;
12
> my $a = False;
False
```

Note that REPL mode always displays a value. If your code prints a value, that is fine. But if not, REPL will print whatever the **last expression** evaluated to.

INSTALLING RAKUDO STAR

Windows & Mac: Use the installation binary. See https://rakudo.org/files/

Linux: Use the normal package system (Debian, Centos, Fedora, openSUSE, Ubuntu and Alpine). See https://nxadm.github.io/rakudo-pkg/

More information: https://perl6.org/downloads/

Or use Docker:
http://perl6maven.com/rakudo-perl6-with-docker

ABOUT PERL6 (PART 1)

VARIABLES & VALUES (PART 1)

PROCEDURES (PART 2)

FILES & IO (PART 2)

VARIABLES

```
> $s = 5;
| ===SORRY!=== Error while compiling:
| Variable '$s' is not declared
| ------> <BOL>⏏$s = 5;
```

Variables must be declared (with my), before they can be used. Or we will get a compile time error, as shown by the polite ===SORRY!===.

The eject symbol (⏏) shows where the compiler thinks the problem is.

```
> my $r = 5;
5
```

SIGILS

The variable type is decided by the first character, called *sigil*. The four types are:

$	Scalar (a single value)
@	Array (several values)
%	Hash (several key-value pairs)
&	Code (callable code)

The sigil may be followed by a «twigil». We will show `*`, indicating a dynamic variable, later.

See https://docs.perl6.org/language/variables#Twigils for more information.

VARIABLE NAMES

The first character (after the sigil and optional twigil) in a variable name (and any other name; e.g. procedures, classes) must be a letter (as in whatever Unicode has decided is a letter) or underscore (_).

The rest can be letters, underscore (_), minus (-),a single quote (') and digits.

A minus (-) or single quote (') must be followed by a letter or underscore (_).

VARIABLE NAMES - EXAMPLES

```
my $r1234;     # OK
my $r1234-56; # ERROR - parsed as "$r1234 - 56"
my $r1234_56; # OK
my $r1234-5A; # ERROR - as "5A" is not a number
my $r1234'5A; # ERROR - as "5A" is not a number
my $Große;     # OK
my $ßßßßßß;    # OK
my $_______;   # OK
my $侮獨;       # OK
```

Common sense is advisable, especially before venturing into the Unicode jungle.

EVERYTHING IS AN OBJECT

If you want it to be.

Most built-in functions have a corresponding method:

```
> say $a;
> $a.say;

> say "Hello";
> "Hello".say;
```

WHAT

The WHAT method can be used to tell us the type of a value or variable:

```
> 12.WHAT;
(Int)

> "12".WHAT;
(Str)

> my $i = 12; $i.WHAT;
(Int)
```

WHAT can be used as a procedure as well: E.g. WHAT 12.

NUMBERS

If it is without quotes, and looks like a number), it is either a number:

```
12        # Integer
12.8      # A number
1.12e+20  # Floating point
2+4i      # Complex
```

Or an error:

```
12A
1.12e
2+4j
```

OCTAL, HEX, BINARY ...

Numbers cannot start with zero, except when specifying the number system:

Octal: `0o123` or `:8<123>`.

Hexadecimal: `0x12A39F` or `:16<12A39F>`.

Binary: `0b10101010` or `:2<10101010>`.

```
> say :2<0b10101010>
170

> say :40<0101000001011111110>
==SORRY!=== Error while compiling:
Radix 40 out of range (allowed: 2..36)
```

UNICODE NUMBERS

Unicode has a lot of characters that are regarded as numeric. Use them if you want to cause confusion:

```
½  # This is a single character.
Ⅷ # A single character (codepoint U+2168)
```

Well. You may not feel that confused, if your terminal or printer support the characters, but what about:

```
> say ৯০৭ + ১; # 907 + 1 (in case you wondered)
908
```

NOT A NUMBER

If it starts with a letter (or an underscore) it is either a procedure call, a predefined value, or an error:

```
print  # -> error: Missing parameter
True   # -> True
False  # -> False
abcdic # -> error: Undeclared routine
```

«True» and «False» are built-in.

N_U_M_B_E_R_S

You can add underscores in numbers to make the code more readable. The compiler ignores them.

```
> my $number1 = 1000000000
1000000000

> my $number2 = 1_000_000_000
1000000000

> $number2 = 1_0_0_0  # This is legal, but stupid.
1000
```

STRINGS

String are specified in quotes; single, double or whatever else Unicode has to offer:

```
> my $name = "Arne"
Arne

> my $hello = "Hello, $name"
Hello, Arne

> my $hello = 'Hello, $name'
Hello, $name

> my $hello = «Hello, $name»
(Hello, Arne)
```

Variables are interpolated, unless single quotes are used.

THE TYPE SYSTEM

Perl6 does not enforce types, unless we ask it to.

```
> my $a;
(Any)

> my Int $i;
(Int)

> $i = "b";
Type check failed in assignment to $i; \
 expected Int but got Str ("b")
```

When a variable does not contain a value, REPL will report the type instead. Any is the most general type, as it can represent anything.

NIL & ANY

Nil is the null value (the absence of a value).

Assign it to a variable to reset it to its default (undefined) value:

```
> my $b = "b"; $b = Nil;
(Any)

> my Int $i = 4; $i = Nil
(Int)
```

It is possible to use the type instead of Nil, but it has no advantage.

CONSTANTS

Do not use variables for values that should stay constant:

```
> constant $pi = 3.14;
```

You cannot change a constant value:

```
> $pi = 3;
Cannot assign to an immutable value
  in block <unit> at <unknown file> line 1
```

Run time errors do not have `===SORRY!===` (as compile time errors do).

FLOATING POINT NUMBERS

Perl6 has several numeric types (in addition to integers, which we have discussed already).

pi is built in:

```
> say pi;
3.14159265358979
> pi.WHAT
(Num)
```

The term «floating point» is derived from the fact that there is no fixed number of digits before and after the decimal point; that is, the decimal point can float. Perl6 calls them «Num».

FLOATING POINT ERRORS

```
> my $one-third = 1/3;
0.333333
```

This is as expected (with the actual number of 3's shown as the only surprise).

Adding three of them should give us 0.999999:

```
> say $one-third * 3;
1
```

But it does not. We do get 1.

RATIONAL NUMBERS

Perl6 has a built in Rat (Rational Number) type.

```
> my $one-third = 1/3;
0.333333

> $one-third.WHAT;
(Rat)

> 0.3.WHAT; # Yes, this is valid syntax!
(Rat)
```

The Rat type is automatically used for values with a decimal fraction, if possible. Otherwise the floating point type Num is used.

RAT IN ACTION

The Rat type uses two integers internally; the actual value is the first divided by the second.

We can use the nude method to get the values:

(«Numerator» + «Denominator»; «Nu» + «De»)

```
> (1/3).nude;
(1 3)

> (0.1).nude;
(1 10)

> 0.2.nude; # Without parens
(1 10)
```

0.333333 ?

So where does 0.333333 come from?

When we **display** any non string value, we get a stringified copy of the value. Perl6 has decided that 6 digits after the decimal point is enough - in this case.

The actual number of digits shown may change.

AUTOMATIC TYPE CONVERSION

```
> my $string1 = "12"; my $string2 = "13";
> my $sum1 = $string1 + $string2;
25
> $sum1.WHAT;
(Int)
> my $sum2 = $string1 ~ $string2; # String concatenation
1213
> $sum2.WHAT;
(Str)
```

Perl6 will automatically convert the values to the required type, if possible.

MANUAL TYPE CONVERSION

Converting a string to a number:

```
> "12".Numeric.WHAT     # -> (Int) # Integer
> "12.1".Numeric.WHAT   # -> (Rat) # Rational number
> "5e+10".Numeric.WHAT  # -> (Num) # Floating point
```

We could have used the `+` prefix instead:

```
> +("12").WHAT    # -> (Int)
> +("12.1").WHAT  # -> (Rat)
```

`Numeric` or the `+` prefix will convert to the best numeric type for the given value.

MANUAL TYPE CONVERSION (2)

Or use one of the types, if you are sure what you want:

```
> "12".Int.WHAT # ->  (Int) # Integer
> "12".Rat.WHAT # ->  (Rat) # Rational number
> "12".Num.WHAT # ->  (Num) # Floating point
> 12.1.Str.WHAT # ->  (Str) # String; "12.1"
> ~(12.1).WHAT  # ->  (Str) # String; "12.1" # Prexif ~
```

But you will get exactly what you ask for:

```
> "12.1".Int # -> 12
```

ARRAYS (AND LISTS)

```
> my @b = "rune", "helge", "tom", "jerry";
[rune helge tom jerry]
```

Add parens, if it makes you feel better...

```
> my @c = ("rune", "helge", "tom", "jerry");
[rune helge tom jerry]
```

Strings must be quoted, as shown above, but we can use a short form if the values do not contain spaces:

```
> my @d = <rune helge tom jerry>;
[rune helge tom jerry]
```

LIST SIZE

```
> my $number-of-elements = @d.elems;
```

Compare with the length of a string:

```
> my $number-of-characters = $s.chars;
```

(There is no «length» method.)

We can specify a size limit:

```
> my @d[10] = <rune helge tom jerry>;
[rune helge tom jerry]

> my @d[3] = <rune helge tom jerry>;
Index 3 for dimension 1 out of range (must be 0..2)
```

LIST ELEMENTS

You can access an individual item by its index:

```
> say @d[0];  # And NOT $d[0] as in perl5!
helge
```

We can access several items (called an array slice):

```
> my @a = 1,2,3,4,5,6,7,8,9,10,11,12,13;
> @a[0 .. 9]; # The same as [0,1,2,3,4,5,6,7,8,9]
(1 2 3 4 5 6 7 8 9 10)

> @a[0,9,2] # They do not need to be consecutive.
(1 10 3)
```

LIST OF LISTS

Perl6 does not automatically flatten lists (as opposed to perl5), so the result of adding (with «push» or «unshift») a second list of items on to a list, is a list with one more item - the second list:

```
> my @list1 = 1,2,3,4,5;
[1 2 3 4 5]

> my @list2 = 1,2;
[1 2]

> @list1.push(@list2);
[1 2 3 4 5 6 7 8 [1 2]]
```

Probably not what you have in mind.

FLATTENING LISTS

If you want to insert the individual values of a second list to a list, use «prepend» (instead of «unshift» and «append» (instead of «push»):

```
> my @list1 = 1,2,3,4,5;
[1 2 3 4 5]

> my @list2 = 8,9;
[8 9]

> @list1.append(@list2); # push the individual values
[1 2 3 4 5 8 9]

> @list1.prepend(@list2); # unshift the individual values
[8 9 1 2 3 4 5 8 9]
```

FLATTENING LISTS (2)

Or you can flatten the list itself, by adding a | (vertical bar) before it:

```
> my @list1 = 1,2,3,4,5;
[1 2 3 4 5]

> my @list2 = 8,9;
[8 9]

> @list1.push(|@list2); # push the individual values
[1 2 3 4 5 8 9]
```

LISTS WITHOUT DUPLICATES

Use unique to get a copy of the list, without duplicates:

```
> (1,1,2,3,4,5,1,6).unique
(1 2 3 4 5 6)
```

If you know that the list is sorted, use squish instead of unique:

```
> (1,1,2,3,4,5,5,6).squish # OK
(1 2 3 4 5 6)

> (1,1,2,3,4,5,1,6).squish # Wrong usage.
(1 2 3 4 5 1 6)
```

LIST ROTATION

List rotations can be done with «push/shift» (to the left) and «unshift/pop» (to the right), but it is easier to use the built in rotate:

```
(1,2,3,4,5,6).rotate;  # Left. The same as rotate(1)
(2 3 4 5 6 1)

> (1,2,3,4,5,6).rotate(2)
(3 4 5 6 1 2)

> (1,2,3,4,5,6).rotate(-2) # Right
(5 6 1 2 3 4)
```

HASHES

```
my %population = (Oslo            => 500_000,
                  Paris           => "unknown",
                  "Buenos Aires" => "too many");
```

The keys (the left hand side of =>) can be specified without quotes - if they do not contain spaces.

We can skip the parens.

Assignment:

```
%population{"Buenos Aires"}  = "too many";
```

And not $population{...} as in perl5.

HASH VALUES

```
> %population{"Oslo"}
500000

> %population<Oslo> # The same
500000

> say %population<Buenos Aires> # An error
((Any) (Any))
```

The last one is the same as:

```
> say ( %population{"Buenos"}, %population{"Aires"} );
((Any) (Any))
```

HASH SLICES

Rember array slices? We can do the same with hashes:

```
> my %translate = ( one => "ein", two => "zwei", \
  three => "drei" );
{one => ein, three => drei, two => zwei}

> say %translate{"two", "one"}
(zwei ein)

> say %translate<two one> # This does not work on arrays.
(zwei ein)
```

Use grep if the selection criteria is more complex:

```
> %translate{%translate.keys.grep(*.chars == 3)}
(zwei ein)
```

HASH LOOKUP

How do we check if a value is present in a hash?

```
> my %h;
> %h<a> = 0;      # => 0
> %h<b> = False; # => False
> %h<c> = Nil;    # => (Any)
```

All of them evaluates to `False` in boolean context.

Use the `:exists` adverb:

```
> %h<a>:exists # => True
> %h<b>:exists # => True
> %h<c>:exists # => True
> %h<d>:exists # => False
```

HASH DELETION

Use the :delete adverb to delete entries from a hash:

```
> my %h = a => 1, b => 2, c => 3
{a => 1, b => 2, c => 3}

> %h<b c>:delete
(2 3)

> %h<a>:delete
1
```

The deleted value or values are returned.

HASH DUPLIATE VALUES

Hashes (obviously) do not allow duplicate values with the same key:

```
> my %hash;
> %hash<M> = 12;
> %hash<M> = "nobody";
> say %hash<M>
nobody
```

HASH DUPLIATE VALUES (2)

But we can get around that by using a list as the value, adding new values to it. We can do this automatically with push:

```
> my %hash;
> %hash<M>.push(12);
> say %hash<M>
[12]
> %hash<M>.push("nobody");
> say %hash<M>
[12 nobody]
> say %hash<M>[1];
nobody
```

HASH DUPLIATE VALUES (3)

But we must do this from the start, as the first push will remove any scalar value already there:

```
> my %hash;
> %hash<M> = 12;
> %hash<M>.push("nobody");
> say %hash<M>
[nobody]
```

HASH KEYS

keys gives (a list of) all the keys:

```
for %population.keys -> $city
{
  say "City $city has %population{$_} people";
}
```

We can use kv (for key-value) instead:

```
for %population.kv -> $city, $population
{
  say "City $city has $population people";
}
```

HASH VALUES

keys gives the keys in random order. If we want order, sort them:

```
> for %population.keys.sort -> $city { ... }
```

values gives all the values:

```
for %population.values -> $population
{
  say "Unknown City with %population{$_} people";
}
```

There is no way to start with a hash value and get back to the key.

ADDITION, FIRST TRY

We can add the values:

```
my $total = 0;
for %population.values -> $population
{
  $total += $population;
}
```

This will give a run time error because of our string values ("unknown" and "too many"):

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

THE «SUM» METHOD

We can use the sum method instead of looping through the values:

```
# File: population-sum (partial)
my $total = %population.values.sum;
```

And again, this fails because of the non-numeric values.

WITH TYPE CHECK

We can check that the value is an Int, before adding it:

```
# File: population-smartmatch (partial)
for %population.values -> $population
{
  $total += $population if $population ~~ Int;
}
```

`~~` is the «smartmatch» operater, which can be used to compare almost anything with almost anyting else.

Here it means «is $population an Int?»

GREP AND TYPE CHECK

We can use grep to get rid of illegal values, before applying sum:

```
my $total = %population.values.grep(* ~~ Int).sum;
```

When the first character in a grep code block is * (a «whatever star») we do not use curlies!

TYPE CONSTRAINTS

Negative integers should be ignored.

A type constraint is the thing, but this is illegal:

```
$total = %population.values.grep(* ~~ Int where * > 0).sum;
```

We can make a new type with «subset», and use that:

```
# File: population-grep (partial)

subset PositiveInt of Int where * > 0;

my $total = %population.values.grep(* ~~ PositiveInt).sum;
```

We could have used a type constraint on the hash (the values): my PositiveInt %population instead, if we got rid of the illogical (and now illegal) values.

THE FULL PROGRAM

```
# File: population-grep

my %population = (Oslo           => 500_000,
                  Xyflwizz       => "unknown",
                  Somewhere      => -100,
                  Paris          => 5_000_000,
                 "Buenos Aires" => 10_000_000);

subset PositiveInt of Int where * > 0;

my $total = %population.values.grep(* ~~ PositiveInt).sum;

say "Total population: $total";
```

END OF PART 1

PART 2

This presentation is available online:
https://npw2018.perl6.eu/