PERL6 IN 45+45 MINUTES

☜ PART 2 ☞

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)

PROCEDURES

Define a procedure like this:

```
sub add ($first-value, $second-value)
{
  return $first-value + $second-value;
}
```

And call it like this:

```
> my $result = add(1, 2);
3
```

Or, if you do not like parens:

```
> my $result = add 1, 2;
3
```

PROCEDURES (2)

What if we try with a text string?

```
> my $result = add "10", 2;
12
```

The string "10" is converted to a number (10), and it works.

It works *because* "10" *can be converted* to a number.

PROCEDURES (3)

But if we try something that cannot be converted, we get a run time error:

```
> my $result = add "ten", 2;
Cannot convert string to number: base-10 number must begin
with valid digits or '.' in '⏏ten' (indicated by ⏏)
  in sub add at <unknown file> line 1
```

TYPE CONSTRAINS

We can use a type constraint to prevent that (and get a compile time error instead):

```
sub add (Numeric $first-value, Numeric $second-value)
{
  return $first-value + $second-value;
}
```

```
> my $result = add "10", 2;

===SORRY!=== Error while compiling:
Calling add(Str, Int) will never work with declared \
 signature (Numeric $first-value, Numeric $second-value)
------> my $result = ⏏add "10", 2;
```

MULTIPLE DISPATCH

We can have different versions of a procedure, with different parameter lists (or «signatures»):

```
multi sub do-something ($file1)          { ... }
multi sub do-something ($file1, $file2) { ... }
```

Or, shorter (without the «sub»):

```
multi do-something ($file-name)      { ... }
multi do-something ($file1, $file2) { ... }
```

With type constraints:

```
multi add (Numeric $value1, Numeric $value2) { ... }
multi add (Str     $value1, Str     $value2) { ... }
```

PROCEDURE ARGUMENTS

Values passed to a procedure are read-only by default:

```
# File: increment
sub increment ($value)
{
  $value++;
  return $value;
}
say increment(12);
```

```
$ perl6 increment
Cannot resolve caller postfix:<++>(Int);
the following candidates match the type
but require mutable arguments:
    (Mu:D $a is rw)
    (Int:D $a is rw)
```

IS RW

The error message gives a hint: is rw («is read write»). This is a trait, that we can add to parameters. Let us try:

```
# File: increment2
sub increment ($value is rw)
{
  $value++;
  return $value;
}
say increment(12);
```

```
$ perl6 increment2
Parameter '$value' expected a writable container,
but got Int value
  in sub increment at increment2 line 3
  in block <unit> at increment2 line 9
```

IS RW (2)

And this fails as well. The problem is that is rw tells the procedure that it can change the variable in the calling code, but we called it with a *value*. And values cannot be changed:

```
> 12 = 13;
Cannot modify an immutable Int (12)
  in block <unit> at <unknown file> line 1
```

IS RW (3)

This works:

```
> my $value  = 12;
> my $result = increment($value);
> say $value  # -> 13
```

But the side effect, that the procedure call changes the value of a variable outside itself without an assignment, is something that should be avoided.

It will fail again if you try to adjust the value passed:

```
> my $result = increment($value + 1);
```

IS COPY

The is copy trait is more fool proof. You get a real variable, with a copy of the value passed to it, and it (the copy) can be changed at will.

```
# File: increment3
sub increment ($value is copy)
{
  $value++;
  return $value;
}
say increment(12);
```

```
$ perl6 increment3
13
```

OPTIONAL ARGUMENTS

It is possible to specify a default value for an argument, making it optional:

```
sub do-something ($value, $optional = "") { ... }
```

We can have more of them:

```
sub do-something-else ($value, $optional1 = 5,
    $optional2 = $value * 2) { ... }
```

It is not possible to assign a value to `$optional2` and not `$optional1`.

```
> do-something-else(11, 101);
```

NAMED ARGUMENTS

A procedure taking many arguments can be a problem. Someone will sooner or later get the order of the arguments wrong.

Named arguments, specified by prefixing the variable with a :, removes that problem, as the order is now irrelevant:

```
> sub aaa (:$a, :$b) { return 2*$a + $b; }
> aaa(a => 2, b => 3);
7
> aaa(b => 3, a => 2);
7
```

NAMED ARGUMENTS (2)

Named arguments makes it possible to have many optional arguments, and you can use as many or few as you want, regardless of order:

```
> sub bbb (:$a = 12, :$b = 13, :$c = 12, :$d = 13 )
> {
>   return $a + $b + $c + $d;
> }
> aaa(a => 1);
> aaa(d => 3, a => 4);
```

NAMED ARGUMENTS (3)

You can mix *normal* (or positional) and named arguments, but the positional ones must come first:

```
> sub ccc ($a, $b, :$c, :$b) { ... }
```

NAMED MANDATORY ARGUMENTS

Use `is required` or the `!` postfix shortform:

```
> sub ccc ($a, $b, :$c!, :$d is required) { ... }
```

Default values are meaningless for mandatory arguments. The compiler will not protest, though. So `:$d is required = False` is legal, even if the default value is useless.

ADVERBS

It is possible to use an alternative *adverb syntax* when specifying named arguments in a procedure call:

```
> aa(a => 1, b => 2);
> aa(:a(1), :b(2));
> aa(:1a,   :2b);

> dd(a => "r", b => "h");
> dd(:a<r>   , :b<h>);
```

Adverbs works with the built-in functions as well.

HELLO, <INSERT NAME HERE>!

Getting input from the command line:

```
file: hello-args

say "Hello, @*ARGS[0]!";
```

@*ARGS is a dynamic variable.

```
> perl6 hello-args NPW
Hello, NPW!
```

HELLO, MAIN

We can use the special MAIN procedure instead of accessing @*ARGS:

```
# File: hello
sub MAIN ($name)
{
  say "Hello, $name!";
}
```

The compiler will execute any code in the program first, and call the MAIN routine afterwards. It is usually not a good idea having any code outside MAIN.

HELLO, ERROR HANDLING

Declare MAIN with as many arguments as you want, with the names you want. The program will fail with a usage message if you give the wrong number (or types) of arguments to the program:

```
$ perl6 hello
Usage:
  hello <name>

$ perl6 hello all
hello, all!

$ perl6 hello all you
Usage:
  hello <name>
```

EVEN BETTER USAGE MESSAGES

The name of the program and the variable name(s) may not say it all. Add a special comment line just above the MAIN procedure(s):

```
# File: hello-usage
#| Person to greet
sub MAIN ($name)
{
  say "Hello, $name!";
}
```

```
$ perl6 hello-usage all you
Usage:
 hello-usage <name> -- Person to greet
```

ABOUT PERL6 (PART 1)

VARIABLES & VALUES (PART 1)

PROCEDURES (PART 2)

FILES & IO (PART 2)

READING A FILE

This program will read a file specified as argument, and print all the lines containing the letter «a»:

```
# File: echo-file-MAIN

sub MAIN ($file-name)
{
  my $fh = open $file-name;

  for $fh.lines -> $line
  {
    say $line if $line.contains("a");
  }
  $fh.close;
}
```

All functions that reads something strips off trailing newline character(s) by default. (And «say» adds them back on.)

IO.LINES

We do not *have to* open (and close) a file explicitly, just to read the content:

```
# File: echo-file-lines

sub MAIN ($file-name)
{
  for $file-name.IO.lines -> $line
  {
    say $line if $line.contains("a");
  }
}
```

LINES IN ONE LINE OF CODE

We can use lines as a procedure without arguments. (We used it as a method in the previous example.) This will read the content of the file(s) specified on the command line:

```
# File: echo-all

.say for lines;
```

No need for MAIN, and it will handle as many files as we want:

```
$ perl6 echo-all /etc/*
```

LINES IN ONE LINE OF CODE (2)

Note that used like this we have no way of getting the file names, or know when one file ends and the next begins.

If invoked without arguments, it will wait for input, and copy it back verbatim. Use <Control-c> to exit.

We will get a warning if one or more files is a directory:

```
'NPW18/' is a directory, cannot do '.open' on a directory
  in block <unit> at echo-all line 3
```

LINES IN ONE LINE OF CODE (3)

We can pipe input to it if we want, and these lines are equal:

```
$ perl6 echo-all file1.txt file2.txt
$ cat file.txt file2.txt | perl6 echo-all
```

ADDING THE "A" FILTER

You may have noticed that I have forgotten the contains("a") filter. As we no longer have a loop, where do we apply an if?

We cannot. But we can use grep:

```
# File: echo-grep

lines.grep({ .contains("a") }).say;
```

The curlies are there to tell grep that we pass it code directly, instead of giving a procedure.

MAIN TAKES SCALARS ONLY

Arguments passed on the command line are always scalars, but we can use a so called slurpy (or «variadic argument») array to grab them all by adding a `*` before the list argument: `*@files`:

```
sub MAIN (*@files)
{
  lines.grep({ .contains("a") }).say;
}
```

We have added `MAIN` for the usage message only. The arguments are ignored.

SLURPY GOTCHA

Note that the slurpy argument allows zero arguments, so this program will behave in the same way as «echo-grep». That means that the usage message will never be triggered (and as such the use of «MAIN» is useless).

We can force it to demand at least one argument:

```
sub MAIN ($file1, *@files) { ... }
```

But good luck explaining the code...

USING A TYPE CONSTRAINT

But using a type constraint gives self-explaining code:

```
# File: echo-grep2

#| One or more files to search for lines with the letter 'a'
sub MAIN (*@files where @files.elems >= 1)
{
  lines.grep({ .contains("a") }).say;
}
```

SLURP

It is possible to read the whole file at once (non-lazily):

```
> my $content = slurp "/home/perl6/bin/echo-file";
```

All strings in perl6 are Unicode, but it is possible to read (and convert) files with other encodings:

```
> my $contents = slurp "/home/arne/echo.c", enc => "latin1";
```

More about supported encodings: https://docs.perl6.org/routine/encoding

FILE CONVERSION

Excercise: Write a file conversion program. Input is in latin1 (iso-latin-1), and output is in Unicode (utf-8).

Hint: Do not try writing to a file (as we have not shown how to do that yet). Writing to the screen (STDOUT) is ok, and we can use the shell to save it for us like this:

```
$ perl6 isolatin2unicode isolatinfile > unicodefile
```

FILE CONVERSION (2)

Solution(s):

```
# File: isolatin2unicode
sub MAIN ($file-name)
{
  say slurp $file-name, enc => "latin1";
}
```

```
# File: isolatin2unicode2
sub MAIN ($file-name)
{
  .say for $file-name.IO.lines(enc => "latin1");
}
```

```
# File: isolatin2unicode3
.say for lines(enc => "latin1");
```

INPUT OUTPUT - IO

On a Unix-like system we have the following predefined filehandles:

$*IN	Standard input filehandle	(STDIN)
$*OUT	Standard output filehandle	(STDOUT)
$*ERR	Standard error filehandle	(STDERR)

PRINT, SAY AND NOTE

print and say prints to the specified filehandle, and to $*OUT if used without one.

note prints to STDERR (the same as $*ERR.say). Do not use it on a filehandle!

WRITING FILES

We can expand the file conversion program to write to a file, if given. The first argument to the program is the file name to read from, as before, and a second one (if given) is the file name to write to.

If we don't specify the second argument, it will print to the screen as before:

```
$ perl6 isolatin2unicode4 isolatinfile unicodefile
$ perl6 isolatin2unicode4 isolatinfile > unicodefile
```

WRITING FILES (2)

We can use a filehandle, open the file in write mode, and use say on the filhandle:

```
> my $fh = open :w, '/tmp/some-file.txt';
> $fh.say "Hello";
> $fh.close;
```

Remember the adverbial syntax; :w is the same as w => True.

But we'll use the spurt command instead. It is the opposite of lines, as it writes all the text we give it to the specified file.

SPURT

```
# File: isolatin2unicode4
sub MAIN ($file-in, $file-out = "")
{
  $file-out
    ?? spurt $file-out, slurp $file-in, enc => "latin1"
    !! say slurp $file-in, enc => "latin1";
}
```

We could have written `$file-out = Nil` instead, but an empty string is ok.

No error checking of any kind, so what can go wrong?

SPURT OVERWRITE

Note that spurt happily overwrites existing files, without warning.

We can instruct spurt to fail if the file exists:

```
spurt $out, :createonly, slurp $in, enc => "latin1";
```

You can add parens if you are confused:

```
spurt($out, :createonly, slurp($in, enc => "latin1"));
```

SPURT APPEND

We also have append mode, that adds the text to the end of an existing file:

```
spurt $file-out, :append, slurp $file-in, enc => "latin1";
```

This is useful when writing to log files.

GET

get reads a single line from the specified filehandle. It returns Nil if no more input is available.

Read one line from standard input:

```
my $line = $*IN.get;
```

Read one line from a file:

```
my $fh = open 'filename';
my $line = $fh.get;
$fh.close;
```

GET GOTCHA

A standalone `get` (without using it on a filehandle) behaves just as `lines`.

It will read from the files given on the command line, and if none are given from `$*IN` instead.

See the special variable `$*ARGFILES` for further information.

PROMPT

prompt is the same as $*IN.get with an optional text output.

```
sub prompt-reimplemented ($message = "")
{
  $*OUT.say $message if $message;
  return $*IN.get;
}
```

```
# File: prompt
my $name = prompt "What's your name? ";
say "Hi, $name! Nice to meet you!";
```

END OF PART 2

Thank you for your attention.

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