Perl 6 hands-on tutorial

A small introduction to a big language

Jonathan Worthington

August 29, 2015

About the course

Not “all of Perl 6”, because we don’t have time.

Rather, enough to help you bootstrap an understanding of Perl 6, so you will be able to go away and keep learning.

Discussion and questions encouraged!

Hands on: Try It And See

Every so often, we have **try it and see** slides: chances for you to try things out for yourself in Perl 6.

They start out simply as one-liners in the REPL; later we move on to writing scripts.

Hopefully you read the session abstract carefully and came prepared. . . ;-)

Pairing encouraged! (Especially those of you who are prepared, with those who are not!)

Orientation

Values

Numerics

We have integers...

```
1
```

And rationals (one integer divided by another):

```
0.25
```

And floating point numbers (the standard IEEE thing):

```
1.23e4
```

Values know their type

You can discover what type a value is using .WHAT. Therefore:

```
say 1.WHAT;
say 0.25.WHAT;
say 1.23e4.WHAT;
```

Tells us:

```
(Int)
(Rat)
(Num)
```

Basic calculations

We have the usual set of numeric operators available, and they are defined on all of the numeric types we've seen so far.

```
say 35 + 7;          # 42
say 1.2 - 0.3;       # 0.9
say 2.1e5 * 3.3e2;   # 69300000
say 168 / 4;         # 42
say 169 % 4;         # 1
say 3 ** 3;          # 27
```

Try it and see!

Your turn! Using the Perl 6 REPL, try and work out answers to the following:

- How big can integers get? (Hint: try raising to a large power!)
- What type do you get when you divide two integers?
- Try $0.1 + 0.2 == 0.3$ in some other language, then try it in Perl 6. What do you observe?

Strings

Perl 6 has lots of quoting constructs. The simplest are single and double quotes.

```
say 'Gangnam style!';
say "Much wow!";
```

The concatenation operator is the tilde. How to remember it? Thing of it as stitching two things together. Also, it looks like a piece of string. :-)

```
say "s" ~ "laughter"; # slaughter
```

Some common string operations

Here are some common operations on strings:

```
say chars("nyanCat");    # 7
say uc("nyanCat");       # NYANCAT
say lc("nyanCat");       # nyancat
say tclc("nyancat");     # Nyancat
```

Alternatively, you might prefer the method forms:

```
say "dogeCoin".chars;    # 8
say "dogeCoin".uc;       # DOGECOIN
say "dogeCoin".lc;       # dogecoin
say "dogeCoin".tclc;     # Dogecoin
```

What does .chars really tell you?

Unicode works at three levels:

- **Graphemes:** Things that a human would think of as "a character"
- **Codepoints:** Things Unicode gives a number to (letters, numbers, marks, diacritics, symbols)
- **Bytes:** the way codepoints are represented on disk, on the wire, etc.

The strings in different programming languages work at different levels. A distinguishing feature of Perl 6 is **strings at grapheme level**.

Therefore, `chars` gives the number of graphemes: the right answer from a human perspective!

But Perl 6 does all the levels. . .

In Perl 6, we have different types representing the different levels you may want to work at:

- `Str` is an immutable string at grapheme level
- `Uni` is an immutable bunch of Unicode codepoints, potentially known to be normalized in some way
- `Buf` is an bunch of bytes (and **Blob** is the same, but immutable)

You can always go between `Str` and `Uni`. However, to go between `Buf` and the higher levels, you must specify an **encoding** (UTF-8, UTF-16, Latin-1, ASCII. . .)

Try it and see!

Take the string:

```
"A\c[COMBINING DOT ABOVE]\c[COMBINING DOT BELOW]"
```

And:

- Use `.chars` to get the number of graphemes, and `.codes` to get the number of Unicode codepoints
- Use `.NFC` and `.NFD` to view composed or decomposed codepoints
- Use `.encode('utf-8')` to see the UTF-8 bytes

If time allows, try using the `uniname` function to get the name of NFC and NFD codepoints. Note that you are shown them in hex, so you'll need to do `uniname(0x41)`, for example.

Booleans

Perl 6 provides `True` and `False` values of type `Bool`.

```
say False.WHAT;     # (Bool)
say True.WHAT;      # (Bool)
```

All types know how to "boolify" themselves. You can ask them to with the `?` operator, or `!` to also negate that boolification.

```
say ?0;     # False
say !0;     # True
say so 0;   # False (lower precedence form of ?)
say not 0;  # True (lower precedence form of !)
```

Coercions

In fact, `?` and `!` are just two **coercion operators**, which move between types. There is also a ~ prefix operator for making something stringy:

```
say (~4.2).WHAT;    # Str
```

And a + prefix operator for making something numeric:

```
say (+"42").WHAT;      # Int
```

Note that trying to numify a string that does not contain a valid number gives a `Failure` (a lazy exception, which is thrown if the value is used, but can be "defused" by testing it; more later!)

Try it and see!

Have a play with coercions. Try to figure out:

- What string values are true and false? `""`? `"0"`? `"False"`?
- What types do you get for `+"42"`, `+"1.5"`, and `+"4.2e4"`? (Hint: be careful with precedence; `.WHAT` binds tighter than a prefix operator.)
- What do True and False numify to?

Variables

Variables

Scalars

A **scalar** variable holds a single item. It has a $ sigil. In Perl 6, we require **explicit declaration** of variables.

```
my $answer = 42;
```

A `Scalar` is actually a first-class object in Perl 6. However, in nearly all operations this is transparent.

```
say $answer.WHAT;   # (Int), not (Scalar)
```

However, it **really is there**. The scope references the `Scalar`, which in turn contains an `Int`.

Assignment vs. binding

Assignment to a Scalar is just putting a new value into it.

```
$answer = 2;            # Store Int 2 in $answer Scalar
$answer = $answer + 19; # Add 19 to $answer and update it
$answer *= 2;           # Short-hand; multiply $answer by 2
```

So far, no surprises. However, Perl 6 also provides **binding**, with the := operator. This stores the bound thing directly in the scope;

```
my $can't-assign-this := "hammer";      # Scope has Str
$can't-assign-this = 'nope, no way';    # Dies
```

.VAR

You won't use binding particularly often, but knowing it exists will help you to understand other parts of the language.

If you're curious, however, you can see whether you have a `Scalar` or not using `.VAR`:

```
my $answer = 42;
say $answer.VAR.WHAT;                   # (Scalar)
my $can't-assign-this := "hammer";
say $can't-assign-this.VAR.WHAT;        # (Int)
```

Note: `.VAR` and `.WHAT` are code smells. They're useful to understand the language and debug, but testing against what they return is nearly always the wrong approach.

Try it and see!

Try to work out the following:

- What does an unassigned Scalar start out containing? (hint: .WHAT)
- Can a variable change its type over time?

Then, try this code:

```
my $a = 42;
my $b := $a;
$b = 100;
say $a;
```

Try to understand what happened.

Arrays

An **array** is an ordered collection of `Scalars`. It can grow on demand, and may also be used like a queue or a stack.

The easiest way to get an array is to declare an `@` variable. When you assign to such a thing, we will parse the right hand side of the assignment at **list precedence**, meaning you can assign without any parentheses:

```
my @countries = 'UK', 'Slovakia', 'Spain', 'Sweden';
```

Arrays are of type `Array`, and know their number of elements:

```
say @countries.WHAT;    # (Array)
say @countries.elems;   # 4
```

Array indexing

The `[...]` array indexer can be used with a single element. Assigning to an element that does not yet exist will extend the array.

```
say @countries[0];                  # UK
@countries[4] = 'Czech Republic';
say @countries.elems;               # 5
```

You can also use multiple indexes. For example, we can swap the order of a couple of elements like this:

```
@countries[1, 2] = @countries[2, 1];
say @countries[0, 1, 2];            # UK Spain Slovakia
say @countries[0..2];               # (the same)
say @countries[^3];                 # (the same)
```

Arrays and binding

When you assign into an array, each element is a `Scalar` container. You can therefore bind array elements.

```
my $first := @countries[0];
$first = 'England';
say @countries[0];                  # England
```

Again, you probably won't do this much yourself. However, it is the mechanism that is behind things like:

```
@values[$idx]++;
```

Quote words

When we initialized our array, we did it as follows:

```
my @countries = 'UK', 'Slovakia', 'Spain', 'Sweden';
```

However, all the quoting gets tedious. We could instead do:

```
my @countries = <UK Slovakia Spain Sweden>;
```

This splits on whitespace. But what if you want to have some elements that contain whitespace? Then you can do this:

```
my @countries =  << "United Kingdom" Slovakia Spain Sweden >>;
```

Stack/queue behaviour

You can `push`/`pop` at the end of an array:

```
my @values;
@values.push(35);
@values.push(7);
@values.push(@values.pop + @values.pop);
```

We could have used the `push`/`pop` functions also.

At the start of an array, we can use `unshift` and `shift`. Using `push` and `shift` in combination, you get queue semantics.

Try it and see!

- What happens if you assign to slot 0 and 3 of an array? What is `.elems`, and what do the slots 1 and 2 contain?
- If you assign one array to another, do you get a fresh set of scalars?
- What happens if you bind an array to another array?
- What happens if you bind a quote-words list directly to an `@variable`? Can you assign to its elements?

Hashes

Hashes are good for storing mappings from one thing to another, when you need to do quick lookups. The easiest way to get a hash is to declare a variable with the % sigil. We typically assign pairs:

```
my %capitals = UK => 'London', Slovakia => 'Bratislava';
```

The variable is of type `Hash`, and the number of elements is the number of key/value pairs:

```
say %capitals.WHAT;     # (Hash)
say %capitals.elems;    # 2
```

Indexing and slicing

Hashes can be indexed with $\{\ldots\}$.

```
say %capitals{'UK'};                   # London
say %capitals{'UK', 'Slovakia'};       # London Bratislava
```

For literal keys, you can use the quote words syntax, which saves some typing and looks neater:

```
say %capitals<UK>;                     # London
say %capitals<UK Slovakia>;            # London Bratislava
```

Existence and deletion

There is no `exists` function or method in Perl 6. Instead, you use the normal indexing syntax and add the `:exists` adverb:

```
say %capitals<UK>:exists;       # True
say %capitals<Elbonia>:exists;  # False
```

The same goes for deletion:

```
%capitals<England> = %capitals<UK>;
%capitals<Scotland> = 'Edinburgh';
%capitals<UK>:delete;
```

Note that `:delete` returns the deleted elements - sometimes handy!

Other useful Hash methods

You can get the keys and values in a hash:

```
my %capitals = UK => 'London', Slovakia => 'Bratislava';
say %capitals.keys;     # UK Slovakia
say %capitals.values;   # London Bratislava
```

You can also get a list of alternating key and value pairs, which - as we'll see later - turns out to be very useful for iteration.

```
say %capitals.kv;       # UK London Slovakia Bratislava
```

Try it and see!

- Are pairs first-class objects in Perl 6? (Hint: `(UK => 'London').WHAT`)
- Suppose you declare the hash in our examples, and have `my $a = 'UK';`. Think about what `%capitals{$a}` will do compared to `%capitals<$a>`. Try it and see if you guessed correctly!
- What happens if you assign one hash to another? What if you assign a hash to an array? How about an array to a hash?
- What happens if you use `.keys`, `.values`, and `.kv` on an array?

Interpolation basics

There are numerous advantages to variables having sigils. One of them is easy **interpolation** of variables in strings.

First off, **single quoted** strings do not interpolate:

```
my $who = 'your mom';
say '$who ate the pie';     # $who ate the pie
```

In double-quoted strings, strings interpolate pretty much as you'd expect:

```
say "$who ate the pie";     # your mom ate the pie
```

Interpolation of arrays and hashes

It's possible to interpolate arrays and hashes, but you need to make a little more effort than with scalars. That way, we don't mangle email addresses.

```
say "Contact me@hotmale.com";   # Contact me@hotmale.com
```

Instead, you need to index. Note that the empty index, known as the **zen slice**, allows you to get all elements.

```
my @winners = <David Ed Nicola>;
say "The winner is @winners[0]";
say "Top 3 were @winners[]";
```

The same rules apply for hashes.

Interpolating method calls

You can also interpolate method calls on any kind of variable:

```
my @winners = <David Ed Nicola>;
say "Top 3 are: @winners.join(', ')";
say "CONGRATULATIONS @winners[0].uc()!";
```

Note that this will not happen if you don't put the parentheses on, so we will not screw up things involving file extensions:

```
my $fh = open "$file.txt";      # Does not try and call .txt
```

Try it and see

- You can embed code inside an interpolated string using curlies `{...}`. Try doing a calculation inside of one - an addition of two variables, say.
- `"<b>$name</b>"` is the one common gotcha from the Perl 6 interpolation rules. What happens if you try it? Why does it happen? Can you find a solution?

Very Basic I/O

The standard handles

Use `say` to write a line to standard output, and `note` to write a line to standard error.

```
say "it again";
note "Uh-oh...";
```

A convenient way to write out a prompt and read a line from standard input is the `prompt` function:

```
my $color = prompt "Name a color: ";
```

Working with files

We can read a file entirely into a scalar using `slurp`, or into an array of lines by getting an IO handle and calling `lines`:

```
my $whole-file = slurp $filename;
my @lines      = $filename.IO.lines;
```

The opposite of `slurp` is `spurt`:

```
spurt $filename, $stuff;
```

Naturally, you can do much more than this - but it's enough for now.

Flow control

Flow control

Comparisons

In Perl 6, we have separate operators for string and numeric comparisons, so even if you don't know the type of the data in the variable, you understand the semantics of the operation.

	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Less than or equal	<=	le
Greater than	>	gt
Greater than or equal	>=	ge

Aside: general design rule

Perl has generally followed the view that different semantics means different operator. This means that you know how the program will behave even if you don't know the exact types of the data.

There's no question that == means numeric equality and eq means string equality, or that + is addition and ~ is concatenation.

In Perl 6 we've de-confused a few things from Perl 5.

```
say 'edam'.flip;          # made
say (1, 2, 3).reverse;    # 3 2 1
say 'omg' x 2;            # (string 'omgomg')
say 'omg' xx 2;           # (a list containing 'omg' twice)
```

if/elsif/else

Basic conditonals are nothing especially surprising. Notice how you don't need any parentheses around the condition.

```
my $num = prompt "Enter a number: ";
if $num < 0 {
    say "Negative";
}
elsif $num > 0 {
    say "Positive";
}
else {
    say "Zero";
}
```

There's also `unless` (which has no `elsif` or `else` equivalent.)

given/when

A given/when is like a switch/case on steroids. We can use it to get the $num variable out of our previous program.

```
given prompt "Enter a number: " {
    when * < 0 {
        say "Negative";
    }
    when * > 0 {
        say "Positive";
    }
    default {
        say "Zero";
    }
}
```

Loops: loop

The `loop` statement is for writing an infinite loop.

```
loop {
    say "Fill all the screen: challenge accepted!";
}
```

You can also use it for C-style loops (but you usually can do better):

```
loop (my $i = 10; $i > 0; $i--) {
    say $i;
}
say "LIFT OFF!";
```

Loops: while/until

The `while` loop is rather unsurprising:

```
my @tasks;
while @tasks.elems < 5 {
    @tasks.push(prompt("Enter a task: "))
}
```

Sometimes, `until` is more natural:

```
my @tasks;
until @tasks.elems == 5 {
    @tasks.push(prompt("Enter a task: "))
}
```

Loops: while/until with pointy blocks

We can also test a value for truth and capture it into a variable only visible inside the block loop:

```
my @tasks;
while prompt("Enter a task (blank if done): ") -> $t {
    @tasks.push($t)
}
```

It works on `if` (and `elsif`) too.

```
if prompt("Enter file to write tasks (optional): ") -> $name {
    # write file
}
```

Try it and see!

Write the classic “guess my number” game. Pick a random number between 1 and 100 (hint: `my $number = (1..100).pick`). Let the user make up to 7 guesses. If they get it right, say they won and `exit;`. If not, state they are too high or too low. If they don’t get it in 7 guesses, tell them they suck.

If you finish this, investigate using `last` instead of `exit`. After this refactor, try to implement giving the option to play again “yes/no”. If you’re still way ahead, make it multi-player, reading in two player names and letting them take turns.

Iteration

The for loop is used to iterate (over arrays, hashes, and so forth). The -> (pointy block) syntax is used again here, to scope the iteration variable cleanly to the block.

```
for @tasks -> $task {
    say $task.tclc;
}
```

Ranges are also common. For example, in the guessing game you could have got the seven guesses using:

```
for ^7 {
    # get/check guess...
}
```

Clarifying ranges

While we're at it, let's just clarify ranges a little more. First off, they are a real object in Perl 6.

```
say (1..10).WHAT;   # Range
```

There are inclusive/exclusive forms of range.

```
1 .. 5      # 1, 2, 3, 4, 5
1 ^.. 5     # 2, 3, 4, 5
1 ..^ 5     # 1, 2, 3, 4
1 ^..^ 5    # 2, 3, 4
^5          # 0, 1, 1, 2, 4 (short for 0..^5)
```

Two at a time

You can take two (or more) items at a time by adding extra variables to the pointy block. This is especially useful on `%hash.kv`:

```
my %capitals = UK => 'London', Slovakia => 'Bratislava';
for %capitals.kv -> $country, $city {
    say "$city is the capital of $country";
}
```

You can do it on an array to loop over the indexes and values together also:

```
for @tasks.kv -> $num, $task {
    say "$num. $task";
}
```

Not just for .kv

You can take two (or three, or more) at a time from any array, not just with `.kv`. For example, suppose we wanted to shuffle and randomly pair up people to dance together, we could do:

```
my @dancers = <Jane Jill Jonny Jimmy Jenny Jack>;
for @dancers.pick(*) -> $first, $second {
    say "$first dances with $second";
}
```

Note that this will fail if there are an odd number of dancers. Soon, we'll discover that the pointy block arguments are just a special case of Perl 6 signatures, and you can make one optional or set a default.

Try it and see!

Create a file with various words in. Loop over the words with a `for` loop (hint: you can put one word per line in the file and use `lines('file'.IO)`). Use a hash to count how many times you've seen each word (use the `++` increment operator). Then loop over the hash to show how many times each word was seen.

If you get done: make it case insensitive, and implement some means to filter out very common words ('a', 'the', 'is', etc').

Subs and signatures

Pointy blocks can go solo

We've already seen that "pointy blocks" can go on `for`/`while` loops, `if` statements, and so forth. However, writing one alone gives you a callable piece of code. You can store it in a variable:

```
my $greeter = -> $greeting, $name { say "$greeting, $name!" };
```

And later call it:

```
$greeter('Hej', 'Carl');    # Hej, Carl!
```

Of course, there's a more convenient way...

Sub declarations

We can use the `sub` keyword to declare a subroutine:

```
sub truncate($text) {
    $text.chars > 100
        ?? "$text.substr(0, 10)..."
        !! $text
}
```

We can then call it in one of the following ways:

```
my $short = truncate($text);    # Function call form
my $short = truncate $text;     # List op form
```

Syntax rules for calls

There are a couple of consistent rules about how calls are parsed.

First, `foo(...)` is **always** a call. It even beats keywords.

```
sub if($cond, $code) { if $cond { $code() } }
if(True, -> { say "Wow, a sub called 'if'!" });
```

Second, whitespace after the sub name will **always** make it a list op. This

```
say substr "craft", 1;      # Correct, passes 2 args to substr
say substr ("craft", 1);    # WRONG, passes 1 arg (a list) to substr
```

Returning results

Our `truncate` sub worked fine because by default a sub returns the value produced by its final statement. We could have been more explicit by using `return`:

```
sub truncate($text) {
    return $text.chars > 100
        ?? "$text.substr(0, 10)..."
        !! $text
}
```

In this case, it's a matter of personal preference. However, in other cases you may need `return` in order to return from a nested block.

Note that **pointy blocks are transparent to `return`**. You can not `return` out of a pointy block, only out of a routine (that is, a `sub` or a `method`, which we'll see soon).

Passing arrays and hashes

Passing arrays and hashes works just like passing scalar values.

```
sub trim-to-shortest(@a, @b) {
    if @a.elems > @b.elems {
        @a[@b.elems .. *]:delete;
    }
    else {
        @b[@a.elems .. *]:delete;
    }
}

my @x = 1..10;
my @y = 5..9;
trim-to-shortest(@x, @y);
say @x;                      # 1 2 3 4 5
```

A note on terminology

In Perl 6, we quite consistently use the terms **argument** and **parameter**.

- We use **argument** to mean the things that you pass when you call a sub (or other piece of code, such as a pointy block)
- We call a set of zero or more arguments an **argument list**
- We use **parameter** to mean the thing you declare that receives the argument
- We call a list of zero or more parameters a **signature**

We try to stick to these in error messages, documentation, and so forth. So it's worth getting clear on them.

Try it and see!

See how a couple of possible failure modes work out:

- What happens if you call a sub with the wrong number of arguments? Do you get an error? Is it compile time or runtime?
- Make a typo when calling a sub. Do we find it at compile time or runtime?

Next, refactor your word frequency program into two subs: `calculate-frequencies` and `display-frequencies`.

Optional parameters

When we put a `?` on a parameter, we make it optional. If no argument is passed for it, then it will contain `Any`. This "undefined" value is false in boolean context, so we can do this:

```
my @dancers = <Jane Jill Jonny Jimmy Jenny>;
for @dancers.pick(*) -> $first, $second? {
    if $second {
        say "$first dances with $second";
    }
    else {
        say "$first dances alone";
    }
}
```

Default values

Another way to make a parameter optional is to supply a **default**:

```
my @dancers = <Jane Jill Jonny Jimmy Jenny>;
for @dancers.pick(*) -> $first, $second = 'the teacher' {
    say "$first dances with $second";
}
```

Defaults are evaluated per call. Further, since we bind left to right, it is possible to default to a previous parameter:

```
sub log-event($event, $occurred = now, $logged = $occurred) {
    # ...
}
```

Named arguments/parameters

It's also possible to use named arguments and parameters. These are typically used for extra configuration and options, and so they are **optional by default**.

```
sub truncate($text, :$limit = 100, :$trailer = '...') {
    $text.chars > $limit
        ?? "$text.substr(0, $limit)$trailer"
        !! $text
}
say truncate("Drink a beer", limit => 11);  # Drink a bee...
```

Other ways to pass named arguments

We have a number of different syntaxes for named arguments.

```
limit => 11      # fat arrow syntax
:limit(11)       # colon pair syntax
```

The colon pair syntax has some special cases for booleans:

```
:global          # same as :global(True)
:!global         # same as :global(False)
```

And also one for passing on a variable with the same name as the parameter:

```
:$global         # same as :global($global)
```

Slurpy parameters

Also we have slurpy parameters, when we want to accept any number of arguments. There are a few forms of these. For example, we can take all positionals:

```
sub join-and-truncate($joiner, $limit, *@values) {
    truncate(@values.join($joiner), limit => $limit);
}
```

Or perhaps also all the named arguments:

```
sub join-and-truncate($joiner, *@values, *%options) {
    # ... but how to pass the options on to truncate?
}
```

Flattening arguments

The “opposite” of slurping is flattening an array or hash into an argument list. This is done with the | prefix operator (which is only meaningful inside of an argument list).

```
sub join-and-truncate($joiner, *@values, *%options) {
    truncate(@values.join($joiner), |%options);
}
```

Slurpy positionals and flattening

It's sometimes important to remember that `*@pos` will flatten incoming arrays into a single array. Therefore, if you want to write this:

```
sub check-array-lengths(*@arrays) {
    for @arrays -> @arr {
        die "Oh no it's too long!" if @arr.elems > 5;
    }
}
my @a = 1..4;
my @b = 1..3;
check-array-lengths(@a, @b);
```

You'll end up in a mess, with `@arrays` containing 7 `Ints`!

Non-flattening slurpy positional

You can prevent this steam-rollering by using `**@arrays` instead:

```
sub check-array-lengths(**@arrays) {
  for @arrays -> @arr {
      die "Oh no it’s too long!" if @arr.elems > 5;
  }
}
my @a = 1..4;
my @b = 1..3;
check-array-lengths(@a, @b);
```

Code arguments

To pass along a subroutine as an argument to another subroutine, pass it by putting the & before its name. On the parameter side, you could happily use a `Scalar`, but using the & sigil there too offers convenient calling.

```
sub run-without-banned-options(&run, %options, @banned) {
    my %filtered = %options;
    %filtered{@banned}:delete;
    run(|%filtered);
}
sub operate(:$force, :$strip, :$verbose) {
    say $force, $strip, $verbose;
}
my %opts = force => 1, verbose => 1;
my @banned = 'force', 'strip';
run-without-banned-options(&operate, %opts, @banned);
```

is copy

Note how we made a copy of the %options hash on the previous slide. We did this so as to avoid modifying the existing one that was passed to us. We can achieve this more easily with is copy:

```
sub run-without-banned-options(&run, %options is copy, @banned) {
    %options{@banned}:delete;
    run(|%options);
}
sub operate(:$force, :$strip, :$verbose) {
    say $force, $strip, $verbose;
}
my %opts = force => 1, verbose => 1;
my @banned = 'force', 'strip';
run-without-banned-options(&operate, %opts, @banned);
```

Try it and see

First, try to write a sub `inc` that, when passed a variable as an argument, adds one to it in place (so `my $a = 41; inc($a); say $a;` gives 42). What happens? Try using `is copy` and see how it changes (but does not fix) the problem. Then try out `is rw`.

Next, make your word frequency filter take the list of blocked words as a named parameter. Also add a `threshold` named parameter (defaulted to zero), and don't return any words with frequencies below the threshold.

Classes, attributes, and methods

What are objects?

“I’m sorry that I long ago coined the term ‘objects’ for this topic because it gets many people to focus on the lesser idea. The big idea is ‘messaging’. . . ” - Alan Kay

Objects are autonomous components with internal state - which we aren’t meant to care about from the outside. Instead, we communicate with an object entirely by sending it messages.

In Perl 6 parlance, as in many other languages, we call message sends “method calls”. But we get better OO designs if we think of them as sending a message to some independent agent.

Declaring a class with attributes

We'll build a simple score keeper for a game where players earn points. Write class to declare a class, and `has` to introduce state private to it. Note the `!` twigil means "this is private to the class".

```
class ScoreKeeper {
    has %!player-points;
}
```

There are various other forms of attribute declaration, but the real name of an attribute is always `$!foo`/`@!foo`/`%!foo`/`&!foo`.

Adding methods

When players earn points, we want to add them. We also want a method to get the standings. Methods look very much like subs, except you declare them with method.

```
method score($player, $points) {
    %!player-points{$player} += $points;
}

method ranking() {
    %!player-points.pairs.sort({ -.value })
}
```

BUILD

Finally, we need to initialize all players to zero points. (Yes, Perl 6 does have auto-vivification, but if we rely on that then our ranking will lack players who never earn points).

We write a `submethod` (a method only callable directly on the class, not inherited) with the special name BUILD to do this:

```
submethod BUILD(:@players) {
    for @players -> $player {
        %!player-points{$player} = 0;
    }
}
```

Creating an instance

We can now create a new instance by calling the new method, which we always inherit. Any named arguments we pass will make it to BUILD:

```
my $game = ScoreKeeper.new(players => <jnthn masak lizmat>);
```

We can then use the object as follows:

```
$game.score('jnthn', 100);
$game.score('lizmat', 150);
for $game.ranking -> $s {
    say "$s.key(): $s.value() points";
}
```

Try it and see!

Write a `DaySchedule` class for scheduling hourly appointments. It should work as follows:

```
my $sched = DaySchedule.new(opening => 9, closing => 17);

$sched.add-appointment(10, 'Fred');     # Works fine
$sched.add-appointment(8, 'Brad');      # Dies: too early
$sched.add-appointment(17, 'Anna');     # Dies: too late
$sched.add-appointment(9, 'Lena');      # Works fine
$sched.add-appointment(10, 'Adam');     # Dies: conflict
```

If you're free time, add a method that describes the schedule.

Roles

A role is a collection of methods, and perhaps attributes, that can be incorporated into a class.

Suppose we wanted to handle point-earning games like those we have so far, along with games where the winner is the first to reach a goal. We can get some re-use by factoring the scoring functionality out into a role.

```
role Scoring {
    has %!player-points;

    method !zero-scores(@players) {
        for @players -> $player {
            %!player-points{$player} = 0;
        }
    }

    method score($player, $points) { ... }
    method ranking() { ... }
}
```

Doing a role

The `does` keyword is used to incorporate a role into a class.

```
class SimpleScoreKeeper does Scoring {
    submethod BUILD(:@players) {
        self!zero-scores(@players);
    }
}
```

You can incorporate many roles into a class. Any **conflicts** - where two roles try to provide a method of the same name - will be identified at compile time.

Re-using Scoring

Here's how we can implement the "first to meet the goal" score class.

```
class FirstToGoalScoreKeeper does Scoring {
    has $!goal;

    submethod BUILD(:@players, :$!goal) {
        self!zero-scores(@players);
    }

    method winners() {
        %!player-points.grep({ .value >= $!goal }).map({ .key })
    }
}
```

If time allows. . .

(whatever topics there's interest in)

Going further

Going further

Where to learn more

The primary source for Perl 6 documentation is **doc.perl6.org**.

There are many examples at **examples.perl6.org**.

There are more links at **perl6.org/documentation**.

Where to ask questions

The **#perl6** channel in **irc.freenode.org** is a great place to get help. Many Perl 6 contributors and users are there, and there is a friendly atmosphere (please help keep it that way!)

If you prefer mailing lists, then the **perl6-users** list is a good place to ask questions also. Find out more at **perl6.org/community/**.

Thank you for coming!

Thank you for coming!