

Perl 6 is a highly capable, feature-rich programming language made for at least the next hundred years.

The primary Perl 6 compiler is called Rakudo, which runs on the JVM and the MoarVM.

Meta-note : the triple pound signs are here to denote headlines, double paragraphs, and single notes.

`#=>` represents the output of a command.

```
# Single line comment start with a pound

#`(  
  Multiline comments use #` and a quoting construct.  
  (), [], {}, , etc, will work.  
)

### Variables

# In Perl 6, you declare a lexical variable using `my`  
my $variable;  
# Perl 6 has 4 kinds of variables:

## * Scalars. They represent a single value. They start with a `$`

my $str = 'String';  
# double quotes allow for interpolation (which we'll see later):  
my $str2 = "String";

# variable names can contain but not end with simple quotes and dashes,  
# and can contain (and end with) underscores :  
# my $weird'variable-name_ = 5; # works !

my $bool = True; # `True` and `False` are Perl 6's boolean  
my $inverse = !$bool; # You can invert a bool with the prefix `!` operator  
my $forced-bool = so $str; # And you can use the prefix `so` operator  
      # which turns its operand into a Bool

## * Lists. They represent multiple values. Their name start with `@`.

my @array = 'a', 'b', 'c';  
# equivalent to :  
my @letters = <a b c>; # array of words, delimited by space.  
      # Similar to perl5's qw, or Ruby's %w.  
my @array = 1, 2, 3;

say @array[2]; # Array indices start at 0 -- This is the third element

say "Interpolate an array using [] : @array[]";  
#=> Interpolate an array using [] : 1 2 3

@array[0] = -1; # Assign a new value to an array index  
@array[0, 1] = 5, 6; # Assign multiple values

my @keys = 0, 2;  
@array[@keys] = @letters; # Assign using an array  
say @array; #=> a 6 b
```

```

## * Hashes, or key-value Pairs.
# Hashes are actually arrays of Pairs
# (you can construct a Pair object using the syntax `Key => Value`),
# except they get "flattened" (hash context), removing duplicated keys.
my %hash = 1 => 2,
           3 => 4;
my %hash = foo => "bar", # keys get auto-quoted
           "some other" => "value", # trailing commas are okay
           ;
my %hash = <key1 value1 key2 value2>; # you can also create a hash
                                     # from an even-numbered array
my %hash = key1 => 'value1', key2 => 'value2'; # same as this

# You can also use the "colon pair" syntax:
# (especially handy for named parameters that you'll see later)
my %hash = :w(1), # equivalent to `w => 1`
           # this is useful for the `True` shortcut:
           :truey, # equivalent to `:truey(True)`, or `truey => True`
           # and for the `False` one:
           :!falsey, # equivalent to `:!falsey(False)`, or `falsey => False`
           ;

say %hash{'key1'}; # You can use {} to get the value from a key
say %hash<key2>;   # If it's a string, you can actually use <>
                  # (`{key1}` doesn't work, as Perl6 doesn't have barewords)

## * Subs (subroutines, or functions in most other languages).
sub say-hello { say "Hello, world" }

sub say-hello-to(Str $name) { # You can provide the type of an argument
                             # and it'll be checked at compile-time.

    say "Hello, $name !";
}

## It can also have optional arguments:
sub with-optional($arg?) { # the "?" marks the argument optional
    say "I might return `(Any)` (Perl's 'null'-like value) if I don't have
        an argument passed, or I'll return my argument";
    $arg;
}
with-optional; # returns Any
with-optional(); # returns Any
with-optional(1); # returns 1

## You can also give them a default value when they're not passed:
sub hello-to($name = "World") {
    say "Hello, $name !";
}
hello-to; #=> Hello, World !
hello-to(); #=> Hello, World !
hello-to('You'); #=> Hello, You !

```

```

## You can also, by using a syntax akin to the one of hashes (yay unified syntax !),
## pass *named* arguments to a `sub`.
# They're optional, and will default to "Any".
sub with-named($normal-arg, :$named) {
    say $normal-arg + $named;
}
with-named(1, named => 6); #=> 7
# There's one gotcha to be aware of, here:
# If you quote your key, Perl 6 won't be able to see it at compile time,
# and you'll have a single Pair object as a positional parameter,
# which means this fails:
with-named(1, 'named' => 6);

with-named(2, :named(5)); #=> 7

# To make a named argument mandatory, you can use `?'s inverse, `!`
sub with-mandatory-named(:$str!) {
    say "$str !";
}
with-mandatory-named(str => "My String"); #=> My String !
with-mandatory-named; # run time error: "Required named parameter not passed"
with-mandatory-named(3); # run time error: "Too many positional parameters passed"

## If a sub takes a named boolean argument ...
sub takes-a-bool($name, :$bool) {
    say "$name takes $bool";
}
# ... you can use the same "short boolean" hash syntax:
takes-a-bool('config', :bool); # config takes True
takes-a-bool('config', :!bool); # config takes False

## You can also provide your named arguments with defaults:
sub named-def(:$def = 5) {
    say $def;
}
named-def; #=> 5
named-def(def => 15); #=> 15

# Since you can omit parenthesis to call a function with no arguments,
# you need "&" in the name to store `say-hello` in a variable.
my &s = &say-hello;
my &other-s = sub { say "Anonymous function !" }

# A sub can have a "slurpy" parameter, or "doesn't-matter-how-many"
sub as-many($head, *@rest) { # `*@` (slurpy) will basically "take everything else".
    # Note: you can have parameters *before* (like here)
    # a slurpy one, but not *after*.
    say @rest.join(' / ') ~ " !";
}
say as-many('Happy', 'Happy', 'Birthday'); #=> Happy / Birthday !
# Note that the splat (the *) did not
# consume the parameter before.

## You can call a function with an array using the

```

```

# "argument list flattening" operator `|`
# (it's not actually the only role of this operator, but it's one of them)
sub concat3($a, $b, $c) {
    say "$a, $b, $c";
}
concat3(|@array); #=> a, b, c
                # `@array` got "flattened" as a part of the argument list

### Containers
# In Perl 6, values are actually stored in "containers".
# The assignment operator asks the container on the left to store the value on
# its right. When passed around, containers are marked as immutable.
# Which means that, in a function, you'll get an error if you try to
# mutate one of your arguments.
# If you really need to, you can ask for a mutable container using `is rw`:
sub mutate($n is rw) {
    $n++;
    say "\$n is now $n !";
}

# If what you want a copy instead, use `is copy`.

# A sub itself returns a container, which means it can be marked as rw:
my $x = 42;
sub x-store() is rw { $x }
x-store() = 52; # in this case, the parentheses are mandatory
               # (else Perl 6 thinks `x-store` is an identifier)
say $x; #=> 52

### Control Flow Structures
## Conditionals

# - `if`
# Before talking about `if`, we need to know which values are "Truthy"
# (represent True), and which are "Falsey" (or "Falsy") -- represent False.
# Only these values are Falsey: 0, (), {}, "", Nil, A type (like `Str` or `Int`),
# and of course False itself.
# Every other value is Truthy.
if True {
    say "It's true !";
}

unless False {
    say "It's not false !";
}

# As you can see, you don't need parentheses around conditions.
# However, you do need the brackets around the "body" block:
# if (true) say; # This doesn't work !

# You can also use their postfix versions, with the keyword after:
say "Quite truthy" if True;

```

```

# - Ternary conditional, "?? !!" (like `x ? y : z` in some other languages)
my $a = $condition ?? $value-if-true !! $value-if-false;

# - `given`-`when` looks like other languages' `switch`, but much more
# powerful thanks to smart matching and thanks to Perl 6's "topic variable", $_.
#
# This variable contains the default argument of a block,
# a loop's current iteration (unless explicitly named), etc.
#
# `given` simply puts its argument into `$_` (like a block would do),
# and `when` compares it using the "smart matching" (`~~`) operator.
#
# Since other Perl 6 constructs use this variable (as said before, like `for`,
# blocks, etc), this means the powerful `when` is not only applicable along with
# a `given`, but instead anywhere a `$_` exists.
given "foo bar" {
    say $_; #=> foo bar
    when /foo/ { # Don't worry about smart matching yet - just know `when` uses it.
        # This is equivalent to `if $_ ~~ /foo/`.
        say "Yay !";
    }
    when $_.chars > 50 { # smart matching anything with True (`$a ~~ True`) is True,
        # so you can also put "normal" conditionals.
        # This when is equivalent to this `if`:
        # if $_ ~~ ($_chars > 50) {...}
        # Which means:
        # if $_.chars > 50 {...}
        say "Quite a long string !";
    }
    default { # same as `when *` (using the Whatever Star)
        say "Something else"
    }
}

## Looping constructs

# - `loop` is an infinite loop if you don't pass it arguments,
# but can also be a C-style `for` loop:
loop {
    say "This is an infinite loop !";
    last; # last breaks out of the loop, like the `break` keyword in other languages
}

loop (my $i = 0; $i < 5; $i++) {
    next if $i == 3; # `next` skips to the next iteration, like `continue`
        # in other languages. Note that you can also use postfix
        # conditionals, loops, etc.
    say "This is a C-style for loop !";
}

# - `for` - Passes through an array
for @array -> $variable {
    say "I've got $variable !";
}

```

```

# As we saw with given, for's default "current iteration" variable is `$_`.
# That means you can use `when` in a `for` just like you were in a `given`.
for @array {
    say "I've got $_";

    .say; # This is also allowed.
        # A dot call with no "topic" (receiver) is sent to `$_` by default
    $_.say; # the above and this are equivalent.
}

for @array {
    # You can...
    next if $_ == 3; # Skip to the next iteration (`continue` in C-like languages).
    redo if $_ == 4; # Re-do the iteration, keeping the same topic variable (`$_`).
    last if $_ == 5; # Or break out of a loop (like `break` in C-like languages).
}

# The "pointy block" syntax isn't specific to for.
# It's just a way to express a block in Perl6.
if long-computation() -> $result {
    say "The result is $result";
}

### Operators

## Since Perl languages are very much operator-based languages,
## Perl 6 operators are actually just funny-looking subroutines, in syntactic
## categories, like infix:<+> (addition) or prefix:<!> (bool not).

## The categories are:
# - "prefix": before (like `!` in `!True`).
# - "postfix": after (like `++` in `$a++`).
# - "infix": in between (like `*` in `4 * 3`).
# - "circumfix": around (like `[ ]` in `[1, 2]`).
# - "post-circumfix": around, after another term (like `{ }` in `%hash{'key'}`).

## The associativity and precedence list are explained below.

# Alright, you're set to go !

## * Equality Checking

# - `==` is numeric comparison
3 == 4; # False
3 != 4; # True

# - `eq` is string comparison
'a' eq 'b';
'a' ne 'b'; # not equal
'a' !eq 'b'; # same as above

# - `eqv` is canonical equivalence (or "deep equality")
(1, 2) eqv (1, 3);

```

```

# - `~~` is smart matching
# For a complete list of combinations, use this table:
# http://perlcabal.org/syn/S03.html#Smart\_matching
'a' ~~ /a/; # true if matches regexp
'key' ~~ %hash; # true if key exists in hash
$arg ~~ &bool-returning-function; # `True` if the function, passed `$arg`
                                   # as an argument, returns `True`.

1 ~~ Int; # "has type" (check superclasses and roles)
1 ~~ True; # smart-matching against a boolean always returns that boolean
            # (and will warn).

# You also, of course, have `<`, `<=`, `>`, `>=`.
# Their string equivalent are also available : `lt`, `le`, `gt`, `ge`.
3 > 4;

## * Range constructors
3 .. 7; # 3 to 7, both included
# `` on either side them exclusive on that side :
3 ^..^ 7; # 3 to 7, not included (basically `4 .. 6`)
# This also works as a shortcut for `0..^N`:
^10; # means 0..^10

# This also allows us to demonstrate that Perl 6 has lazy/infinite arrays,
# using the Whatever Star:
my @array = 1..*; # 1 to Infinite ! `1..Inf` is the same.
say @array[^10]; # you can pass arrays as subscripts and it'll return
                 # an array of results. This will print
                 # "1 2 3 4 5 6 7 8 9 10" (and not run out of memory !)
# Note : when reading an infinite list, Perl 6 will "reify" the elements
# it needs, then keep them in memory. They won't be calculated more than once.
# It also will never calculate more elements that are needed.

# An array subscript can also be a closure.
# It'll be called with the length as the argument
say join(' ', @array[15..*]); #=> 15 16 17 18 19
# which is equivalent to:
say join(' ', @array[-> $n { 15..$n }]);
# Note: if you try to do either of those with an infinite array,
#       you'll trigger an infinite loop (your program won't finish)

# You can use that in most places you'd expect, even assigning to an array
my @numbers = ^20;

# Here numbers increase by "6"; more on `...` operator later.
my @seq = 3, 9 ... * > 95; # 3 9 15 21 27 [...] 81 87 93 99;
@numbers[5..*] = 3, 9 ... *; # even though the sequence is infinite,
                             # only the 15 needed values will be calculated.
say @numbers; #=> 0 1 2 3 4 3 9 15 21 [...] 81 87
              # (only 20 values)

## * And, Or
3 && 4; # 4, which is Truthy. Calls `.Bool` on `4` and gets `True`.
0 || False; # False. Calls `.Bool` on `0`

```

```

## * Short-circuit (and tight) versions of the above
$a && $b && $c; # Returns the first argument that evaluates to False,
                # or the last argument.

$a || $b;

# And because you're going to want them,
# you also have compound assignment operators:
$a *= 2; # multiply and assignment
$b %%= 5; # divisible by and assignment
@array .= sort; # calls the `sort` method and assigns the result back

### More on subs !
# As we said before, Perl 6 has really powerful subs. We're going to see
# a few more key concepts that make them better than in any other language :-).

## Unpacking !
# It's the ability to "extract" arrays and keys (AKA "destructuring").
# It'll work in `my`s and in parameter lists.
my ($a, $b) = 1, 2;
say $a; #=> 1
my ($, $, $c) = 1, 2, 3; # keep the non-interesting anonymous
say $c; #=> 3

my ($head, *@tail) = 1, 2, 3; # Yes, it's the same as with "slurpy subs"
my (*@small) = 1;

sub foo(@array [$fst, $snd]) {
    say "My first is $fst, my second is $snd ! All in all, I'm @array[].";
    # (^ remember the `[]` to interpolate the array)
}
foo(@tail); #=> My first is 2, my second is 3 ! All in all, I'm 2 3

# If you're not using the array itself, you can also keep it anonymous,
# much like a scalar:
sub first-of-array(@ [$fst]) { $fst }
first-of-array(@small); #=> 1
first-of-array(@tail); # Throws an error "Too many positional parameters passed"
                       # (which means the array is too big).

# You can also use a slurp ...
sub slurp-in-array(@ [$fst, *@rest]) { # You could keep `*@rest` anonymous
    say $fst + @rest.elems; # `.elems` returns a list's length.
    # Here, `@rest` is `(3,)`, since `$fst` holds the `2`.
}
slurp-in-array(@tail); #=> 3

# You could even extract on a slurpy (but it's pretty useless ;-).)
sub fst(*@ [$fst]) { # or simply : `sub fst($fst) { ... }`
    say $fst;
}
fst(1); #=> 1
fst(1, 2); # errors with "Too many positional parameters passed"

```



```

# You can also destructure hashes (and classes, which you'll learn about later !)
# The syntax is basically `%hash-name (:key($variable-to-store-value-in))`.
# The hash can stay anonymous if you only need the values you extracted.
sub key-of(% (:value($val), :qua($qua))) {
  say "Got val $val, $qua times.";
}

# Then call it with a hash: (you need to keep the brackets for it to be a hash)
key-of({value => 'foo', qua => 1});
#key-of(%hash); # the same (for an equivalent `%hash`)

## The last expression of a sub is returned automatically
# (though you may use the `return` keyword, of course):
sub next-index($n) {
  $n + 1;
}
my $new-n = next-index(3); # $new-n is now 4

# This is true for everything, except for the looping constructs
# (due to performance reasons): there's reason to build a list
# if we're just going to discard all the results.
# If you still want to build one, you can use the `do` statement prefix:
# (or the `gather` prefix, which we'll see later)
sub list-of($n) {
  do for ^$n { # note the use of the range-to prefix operator `^` (`0..^N`)
    $_ # current loop iteration
  }
}
my @list3 = list-of(3); #=> (0, 1, 2)

## You can create a lambda with `-> {}` ("pointy block") or `{}` ("block")
my &lambda = -> $argument { "The argument passed to this lambda is $argument" }
# `-> {}` and `{}` are pretty much the same thing, except that the former can
# take arguments, and that the latter can be mistaken as a hash by the parser.

# We can, for example, add 3 to each value of an array using map:
my @arrayplus3 = map({ $_ + 3 }, @array); # $_ is the implicit argument

# A sub (`sub {}`) has different semantics than a block (`{}` or `-> {}`):
# A block doesn't have a "function context" (though it can have arguments),
# which means that if you return from it,
# you're going to return from the parent function. Compare:
sub is-in(@array, $elem) {
  # this will `return` out of the `is-in` sub
  # once the condition evaluated to True, the loop won't be run anymore
  map({ return True if $_ == $elem }, @array);
}
sub truthy-array(@array) {
  # this will produce an array of `True` and `False`:
  # (you can also say `anon sub` for "anonymous subroutine")
  map(sub ($i) { if $i { return True } else { return False } }, @array);
  # ~ the `return` only returns from the anonymous `sub`
}

```

```

# You can also use the "whatever star" to create an anonymous function
# (it'll stop at the furthest operator in the current expression)
my @arrayplus3 = map(*+3, @array); # `*+3` is the same as `{ $_ + 3 }`
my @arrayplus3 = map(+++3, @array); # Same as `-> $a, $b { $a + $b + 3 }`
                                   # also `sub ($a, $b) { $a + $b + 3 }`

say (*~2)(4); #=> 2
           # Immediately execute the function Whatever created.
say ((*+3)/5)(5); #=> 1.6
           # works even in parens !

# But if you need to have more than one argument (`$_`)
# in a block (without wanting to resort to `-> {}`),
# you can also use the implicit argument syntax, `$^` :
map({ $^a + $^b + 3 }, @array); # equivalent to following:
map(sub ($a, $b) { $a + $b + 3 }, @array); # (here with `sub`)

# Note : those are sorted lexicographically.
# `{ $b / $a }` is like `-> $a, $b { $b / $a }`

## About types...
# Perl6 is gradually typed. This means you can specify the type
# of your variables/arguments/return types, or you can omit them
# and they'll default to "Any".
# You obviously get access to a few base types, like Int and Str.
# The constructs for declaring types are "class", "role",
# which you'll see later.

# For now, let us examine "subset":
# a "subset" is a "sub-type" with additional checks.
# For example: "a very big integer is an Int that's greater than 500"
# You can specify the type you're subtyping (by default, Any),
# and add additional checks with the "where" keyword:
subset VeryBigInteger of Int where * > 500;

## Multiple Dispatch
# Perl 6 can decide which variant of a `sub` to call based on the type of the
# arguments, or on arbitrary preconditions, like with a type or a `where`:

# with types
multi sub sayit(Int $n) { # note the `multi` keyword here
    say "Number: $n";
}
multi sayit(Str $s) { # a multi is a `sub` by default
    say "String: $s";
}
sayit("foo"); # prints "String: foo"
sayit(True); # fails at *compile time* with
              # "calling 'sayit' will never work with arguments of types ..."

# with arbitrary precondition (remember subsets?):
multi is-big(Int $n where * > 50) { "Yes !" } # using a closure
multi is-big(Int $ where 10..50) { "Quite." } # Using smart-matching
                                              # (could use a regexp, etc)

```

```

multi is-big(Int $) { "No" }

subset Even of Int where * %% 2;

multi odd-or-even(Even) { "Even" } # The main case using the type.
                                   # We don't name the argument.
multi odd-or-even($) { "Odd" } # "else"

# You can even dispatch based on a positional's argument presence !
multi with-or-without-you(:$with!) { # You need make it mandatory to
                                   # be able to dispatch against it.
    say "I can live ! Actually, I can't.";
}
multi with-or-without-you {
    say "Definitely can't live.";
}
# This is very, very useful for many purposes, like `MAIN` subs (covered later),
# and even the language itself is using it in several places.
#
# - `is`, for example, is actually a `multi sub` named `trait_mod:<is>`,
# and it works off that.
# - `is rw`, is simply a dispatch to a function with this signature:
# sub trait_mod:<is>(Routine $r, :$rw!) {}
#
# (commented because running this would be a terrible idea !)

### Scoping
# In Perl 6, contrarily to many scripting languages (like Python, Ruby, PHP),
# you are to declare your variables before using them. You know `my`.
# (there are other declarators, `our`, `state`, ..., which we'll see later).
# This is called "lexical scoping", where in inner blocks,
# you can access variables from outer blocks.
my $foo = 'Foo';
sub foo {
    my $bar = 'Bar';
    sub bar {
        say "$foo $bar";
    }
    &bar; # return the function
}
foo(); #=> 'Foo Bar'

# As you can see, `$foo` and `$bar` were captured.
# But if we were to try and use `$bar` outside of `foo`,
# the variable would be undefined (and you'd get a compile time error).

# Perl 6 has another kind of scope : dynamic scope.
# They use the twigil (composed sigil) `*` to mark dynamically-scoped variables:
my $*a = 1;
# Dynamically-scoped variables depend on the current call stack,
# instead of the current block depth.
sub foo {
    my $*foo = 1;

```

```

    bar(); # call `bar` in-place
}
sub bar {
    say $*foo; # `$*foo` will be looked in the call stack, and find `foo`'s,
                # even though the blocks aren't nested (they're call-nested).
                #=> 1
}

### Object Model

# You declare a class with the keyword `class`, fields with `has`,
# methods with `method`. Every attribute that is private is named `${!attr}`.
# Immutable public attributes are named `${attr}`
# (you can make them mutable with `is rw`)

# Perl 6's object model ("SixModel") is very flexible,
# and allows you to dynamically add methods, change semantics, etc ...
# (this will not be covered here, and you should refer to the Synopsis).

class A {
    has `${.field}`; # ``${.field}` is immutable.
                    # From inside the class, use `${!field}` to modify it.
    has `${.other-field}` is rw; # You can mark a public attribute `rw`.
    has Int `${!private-field}` = 10;

    method get-value {
        `${.field}` + `${!private-field}`;
    }

    method set-value(`${$n}`) {
        # `${.field}` = `${$n}`; # As stated before, you can't use the `${.}` immutable version.
        `${!field}` = `${$n}`; # This works, because `${!}` is always mutable.

        `${.other-field}` = 5; # This works, because `${.other-field}` is `rw`.
    }

    method `${!private-method}` {
        say "This method is private to the class !";
    }
};

# Create a new instance of A with `${.field}` set to 5 :
# Note: you can't set private-field from here (more later on).
my `${$a}` = A.new(field => 5);
`${$a}.get-value; #=> 15
`${$a}.field` = 5; # This fails, because the `has `${.field}`` is immutable
`${$a}.other-field` = 10; # This, however, works, because the public field
                        # is mutable (`rw`).

## Perl 6 also has inheritance (along with multiple inheritance)

class A {
    has `${.val}`;

```

```

submethod not-inherited {
  say "This method won't be available on B.";
  say "This is most useful for BUILD, which we'll see later";
}

method bar { $.val * 5 }
}
class B is A { # inheritance uses `is`
  method foo {
    say $.val;
  }

  method bar { $.val * 10 } # this shadows A's `bar`
}

# When you use `my T $var`, `$var` starts off with `T` itself in it,
# so you can call `new` on it.
# (`.=` is just the dot-call and the assignment operator:
# `$a .= b` is the same as `$a = $a.b`)
# Also note that `BUILD` (the method called inside `new`)
# will set parent properties too, so you can pass `val => 5`.
my B $b .= new(val => 5);

# $b.not-inherited; # This won't work, for reasons explained above
$b.foo; # prints 5
$b.bar; #=> 50, since it calls B's `bar`

## Roles are supported too (also called Mixins in other languages)
role PrintableVal {
  has $!counter = 0;
  method print {
    say $.val;
  }
}

# you "import" a mixin (a "role") with "does":
class Item does PrintableVal {
  has $.val;

  # When `does`-ed, a `role` literally "mixes in" the class:
  # the methods and fields are put together, which means a class can access
  # the private fields/methods of its roles (but not the inverse !):
  method access {
    say $!counter++;
  }

  # However, this:
  # method print {}
  # is ONLY valid when `print` isn't a `multi` with the same dispatch.
  # (this means a parent class can shadow a child class's `multi print() {}`,
  # but it's an error if a role does)

  # NOTE: You can use a role as a class (with `is ROLE`). In this case, methods
  # will be shadowed, since the compiler will consider `ROLE` to be a class.

```

```

}

### Exceptions
# Exceptions are built on top of classes, in the package `X` (like `X::IO`).
# Unlike many other languages, in Perl 6, you put the `CATCH` block within the
# block to `try`. By default, a `try` has a `CATCH` block that catches
# any exception (`CATCH { default {} }`).
# You can redefine it using `when`s (and `default`)
# to handle the exceptions you want:
try {
  open 'foo';
  CATCH {
    when X::AdHoc { say "unable to open file !" }
    # Any other exception will be re-raised, since we don't have a `default`
    # Basically, if a `when` matches (or there's a `default`) marks the exception as
    # "handled" so that it doesn't get re-thrown from the `CATCH`.
    # You still can re-throw the exception (see below) by hand.
  }
}

# You can throw an exception using `die`:
die X::AdHoc.new(payload => 'Error !');

# You can access the last exception with `$!` (use `$ _` in a `CATCH` block)

# There are also some subtelties to exceptions. Some Perl 6 subs return a `Failure`,
# which is a kind of "unthrown exception". They're not thrown until you tried to look
# at their content, unless you call `.Bool`/`.defined` on them - then they're handled.
# (the `.handled` method is `rw`, so you can mark it as `False` back yourself)
#
# You can throw a `Failure` using `fail`. Note that if the pragma `use fatal` is on,
# `fail` will throw an exception (like `die`).
fail "foo"; # We're not trying to access the value, so no problem.
try {
  fail "foo";
  CATCH {
    default { say "It threw because we tried to get the fail's value!" }
  }
}

# There is also another kind of exception: Control exceptions.
# Those are "good" exceptions, which happen when you change your program's flow,
# using operators like `return`, `next` or `last`.
# You can "catch" those with `CONTROL` (not 100% working in Rakudo yet).

### Packages
# Packages are a way to reuse code. Packages are like "namespaces", and any
# element of the six model (`module`, `role`, `class`, `grammar`, `subset`
# and `enum`) are actually packages. (Packages are the lowest common denominator)
# Packages are important - especially as Perl is well-known for CPAN,
# the Comprehensive Perl Archive Network.
# You're not supposed to use the package keyword, usually:
# you use `class Package::Name::Here;` to declare a class,
# or if you only want to export variables/subs, you can use `module`:

```

```

module Hello::World { # Bracketed form
    # If `Hello` doesn't exist yet, it'll just be a "stub",
    # that can be redeclared as something else later.
    # ... declarations here ...
}
unit module Parse::Text; # file-scoped form
grammar Parse::Text::Grammar { # A grammar is a package, which you could `use`
}

# You can use a module (bring its declarations into scope) with `use`
use JSON::Tiny; # if you installed Rakudo* or Panda, you'll have this module
say from-json('[1]').perl; #=> [1]

# As said before, any part of the six model is also a package.
# Since `JSON::Tiny` uses (its own) `JSON::Tiny::Actions` class, you can use it:
my $actions = JSON::Tiny::Actions.new;

# We'll see how to export variables and subs in the next part:

### Declarators
# In Perl 6, you get different behaviors based on how you declare a variable.
# You've already seen `my` and `has`, we'll now explore the others.

## * `our` (happens at `INIT` time -- see "Phasers" below)
# It's like `my`, but it also creates a package variable.
# (All packagish things (`class`, `role`, etc) are `our` by default)
module Foo::Bar {
    our $n = 1; # note: you can't put a type constraint on an `our` variable
    our sub inc {
        our sub available { # If you try to make inner `sub`s `our`...
            # Better know what you're doing (Don't !).
            say "Don't do that. Seriously. You'd get burned.";
        }
        my sub unavailable { # `my sub` is the default
            say "Can't access me from outside, I'm my !";
        }
        say ++$n; # increment the package variable and output its value
    }
}
say $Foo::Bar::n; #=> 1
Foo::Bar::inc; #=> 2
Foo::Bar::inc; #=> 3

## * `constant` (happens at `BEGIN` time)
# You can use the `constant` keyword to declare a compile-time variable/symbol:
constant Pi = 3.14;
constant $var = 1;

# And if you're wondering, yes, it can also contain infinite lists.
constant why-not = 5, 15 ... *;
say why-not[~5]; #=> 5 15 25 35 45

## * `state` (happens at run time, but only once)
# State variables are only initialized one time

```

```

# (they exist in other languages such as C as `static`)
sub fixed-rand {
    state $val = rand;
    say $rand;
}
fixed-rand for ^10; # will print the same number 10 times

# Note, however, that they exist separately in different enclosing contexts.
# If you declare a function with a `state` within a loop, it'll re-create the
# variable for each iteration of the loop. See:
for ^5 -> $a {
    sub foo {
        state $val = rand; # This will be a different value for every value of `$a`
    }
    for ^5 -> $b {
        say foo; # This will print the same value 5 times, but only 5.
                # Next iteration will re-run `rand`.
    }
}

### Phasers
# Phasers in Perl 6 are blocks that happen at determined points of time in your
# program. When the program is compiled, when a for loop runs, when you leave a
# block, when an exception gets thrown ... (`CATCH` is actually a phaser !)
# Some of them can be used for their return values, some of them can't
# (those that can have a "[" in the beginning of their explanation text).
# Let's have a look !

## * Compile-time phasers
BEGIN { say "[" Runs at compile time, as soon as possible, only once" }
CHECK { say "[" Runs at compile time, as late as possible, only once" }

## * Run-time phasers
INIT { say "[" Runs at run time, as soon as possible, only once" }
END { say "Runs at run time, as late as possible, only once" }

## * Block phasers
ENTER { say "[" Runs everytime you enter a block, repeats on loop blocks" }
LEAVE { say "Runs everytime you leave a block, even when an exception
    happened. Repeats on loop blocks." }

PRE { say "Asserts a precondition at every block entry,
    before ENTER (especially useful for loops)" }
# exemple:
for 0..2 {
    PRE { $_ > 1 } # This is going to blow up with "Precondition failed"
}

POST { say "Asserts a postcondition at every block exit,
    after LEAVE (especially useful for loops)" }
for 0..2 {
    POST { $_ < 2 } # This is going to blow up with "Postcondition failed"
}

```



```

}

## * Block/exceptions phasers
sub {
    KEEP { say "Runs when you exit a block successfully (without throwing an exception)" }
    UNDO { say "Runs when you exit a block unsuccessfully (by throwing an exception)" }
}

## * Loop phasers
for ^5 {
    FIRST { say "[*] The first time the loop is run, before ENTER" }
    NEXT { say "At loop continuation time, before LEAVE" }
    LAST { say "At loop termination time, after LEAVE" }
}

## * Role/class phasers
COMPOSE { "When a role is composed into a class. /\ NOT YET IMPLEMENTED" }

# They allow for cute tricks or clever code ...:
say "This code took " ~ (time - CHECK time) ~ "s to compile";

# ... or clever organization:
sub do-db-stuff {
    $db.start-transaction; # start a new transaction
    KEEP $db.commit; # commit the transaction if all went well
    UNDO $db.rollback; # or rollback if all hell broke loose
}

### Statement prefixes
# Those act a bit like phasers: they affect the behavior of the following code.
# Though, they run in-line with the executable code, so they're in lowercase.
# (`try` and `start` are theoretically in that list, but explained somewhere else)
# Note: all of these (except start) don't need explicit brackets `{` and `}`.

# - `do` (that you already saw) - runs a block or a statement as a term
# You can't normally use a statement as a value (or "term"):
#
#   my $value = if True { 1 } # `if` is a statement - parse error
#
# This works:
my $a = do if True { 5 } # with `do`, `if` is now a term.

# - `once` - Makes sure a piece of code only runs once
for ^5 { once say 1 }; #=> 1
# Only prints ... once.
# Like `state`, they're cloned per-scope
for ^5 { sub { once say 1 }() } #=> 1 1 1 1 1
# Prints once per lexical scope

# - `gather` - Co-routine thread
# Gather allows you to `take` several values in an array,
# much like `do`, but allows you to take any expression.
say gather for ^5 {
    take $_ * 3 - 1;
}

```

```

    take $_ * 3 + 1;
} #=> -1 1 2 4 5 7 8 10 11 13
say join ',', gather if False {
    take 1;
    take 2;
    take 3;
} # Doesn't print anything.

# - `eager` - Evaluate statement eagerly (forces eager context)
# Don't try this at home:
#
#     eager 1..*; # this will probably hang for a while (and might crash ...).
#
# But consider:
constant thrice = gather for ^3 { say take $_ }; # Doesn't print anything
# versus:
constant thrice = eager gather for ^3 { say take $_ }; #=> 0 1 2

# - `lazy` - Defer actual evaluation until value is fetched (forces lazy context)
# Not yet implemented !!

# - `sink` - An `eager` that discards the results (forces sink context)
constant nilthingie = sink for ^3 { .say } #=> 0 1 2
say nilthingie.perl; #=> Nil

# - `quietly` - Suppresses warnings
# Not yet implemented !

# - `contend` - Attempts side effects under STM
# Not yet implemented !

### More operators thingies !

## Everybody loves operators ! Let's get more of them

# The precedence list can be found here:
# http://perlcabal.org/syn/S03.html#Operator\_precedence
# But first, we need a little explanation about associativity:

# * Binary operators:
$a ! $b ! $c; # with a left-associative `!`, this is `($a ! $b) ! $c`
$a ! $b ! $c; # with a right-associative `!`, this is `$a ! ($b ! $c)`
$a ! $b ! $c; # with a non-associative `!`, this is illegal
$a ! $b ! $c; # with a chain-associative `!`, this is `($a ! $b) and ($b ! $c)`
$a ! $b ! $c; # with a list-associative `!`, this is `infix:<>`

# * Unary operators:
!$a! # with left-associative `!`, this is `(!$a)!`
!$a! # with right-associative `!`, this is `!(($a!))`
!$a! # with non-associative `!`, this is illegal

## Create your own operators !
# Okay, you've been reading all of that, so I guess I should try
# to show you something exciting.

```

```

# I'll tell you a little secret (or not-so-secret):
# In Perl 6, all operators are actually just funny-looking subroutines.

# You can declare an operator just like you declare a sub:
sub prefix:<win>($winner) { # refer to the operator categories
    # (yes, it's the "words operator" `<>`)

    say "$winner Won !";
}
win "The King"; #=> The King Won !
                # (prefix is before)

# you can still call the sub with its "full name"
say prefix:<!>(True); #=> False

sub postfix:<!>(Int $n) {
    [*] 2..$n; # using the reduce meta-operator ... See below ;- ) !
}
say 5!; #=> 120
        # Postfix operators (after) have to come *directly* after the term.
        # No whitespace. You can use parentheses to disambiguate, i.e. `(5!)!`

sub infix:<times>(Int $n, Block $r) { # infix in the middle
    for ^$n {
        $r(); # You need the explicit parentheses to call the function in `$r`,
              # else you'd be referring at the variable itself, like with `@r`.
    }
}
3 times -> { say "hello" }; #=> hello
                        #=> hello
                        #=> hello
                        # You're very recommended to put spaces
                        # around your infix operator calls.

# For circumfix and post-circumfix ones
sub circumfix:<[ ]>(Int $n) {
    $n ** $n
}
say [5]; #=> 3125
        # circumfix is around. Again, no whitespace.

sub postcircumfix:<{ }>(Str $s, Int $idx) {
    # post-circumfix is
    # "after a term, around something"
    $s.substr($idx, 1);
}
say "abc">{1}; #=> b
                # after the term `"abc"`, and around the index (1)

# This really means a lot -- because everything in Perl 6 uses this.
# For example, to delete a key from a hash, you use the `:delete` adverb
# (a simple named argument underneath):
%h{$key}:delete;
# equivalent to:

```

```

postcircumfix:<{ }>(%h, $key, :delete); # (you can call operators like that)
# It's *all* using the same building blocks!
# Syntactic categories (prefix infix ...), named arguments (adverbs), ...,
# - used to build the language - are available to you.

# (you are, obviously, recommended against making an operator out of
# *everything* -- with great power comes great responsibility)

## Meta operators !
# Oh boy, get ready. Get ready, because we're delving deep
# into the rabbit's hole, and you probably won't want to go
# back to other languages after reading that.
# (I'm guessing you don't want to already at that point).
# Meta-operators, as their name suggests, are *composed* operators.
# Basically, they're operators that apply another operator.

## * Reduce meta-operator
# It's a prefix meta-operator that takes a binary function and
# one or many lists. If it doesn't get passed any argument,
# it either returns a "default value" for this operator
# (a meaningless value) or `Any` if there's none (examples below).
#
# Otherwise, it pops an element from the list(s) one at a time, and applies
# the binary function to the last result (or the list's first element)
# and the popped element.
#
# To sum a list, you could use the reduce meta-operator with `+`, i.e.:
say [+] 1, 2, 3; #=> 6
# equivalent to `(1+2)+3`
say [*] 1..5; #=> 120
# equivalent to `((((1*2)*3)*4)*5)`

# You can reduce with any operator, not just with mathematical ones.
# For example, you could reduce with `//` to get
# the first defined element of a list:
say [//] Nil, Any, False, 1, 5; #=> False
# (Falsey, but still defined)

# Default value examples:
say [*] (); #=> 1
say [+] (); #=> 0
# meaningless values, since  $N*1=N$  and  $N+0=N$ .
say [//]; #=> (Any)
# There's no "default value" for `//`.

# You can also call it with a function you made up, using double brackets:
sub add($a, $b) { $a + $b }
say [[&add]] 1, 2, 3; #=> 6

## * Zip meta-operator
# This one is an infix meta-operator than also can be used as a "normal" operator.
# It takes an optional binary function (by default, it just creates a pair),
# and will pop one value off of each array and call its binary function on these

```

```

# until it runs out of elements. It returns an array with all of these new elements.
(1, 2) Z (3, 4); # ((1, 3), (2, 4)), since by default, the function makes an array
1..3 Z+ 4..6; # (5, 7, 9), using the custom infix:<+> function

# Since `Z` is list-associative (see the list above),
# you can use it on more than one list
(True, False) Z|| (False, False) Z|| (False, False); # (True, False)

# And, as it turns out, you can also use the reduce meta-operator with it:
[Z||] (True, False), (False, False), (False, False); # (True, False)

## And to end the operator list:

## * Sequence operator
# The sequence operator is one of Perl 6's most powerful features:
# it's composed of first, on the left, the list you want Perl 6 to deduce from
# (and might include a closure), and on the right, a value or the predicate
# that says when to stop (or Whatever for a lazy infinite list).
my @list = 1, 2, 3 ... 10; # basic deducing
#my @list = 1, 3, 6 ... 10; # this dies because Perl 6 can't figure out the end
my @list = 1, 2, 3 ...^ 10; # as with ranges, you can exclude the last element
# (the iteration when the predicate matches).
my @list = 1, 3, 9 ... * > 30; # you can use a predicate
# (with the Whatever Star, here).
my @list = 1, 3, 9 ... { $_ > 30 }; # (equivalent to the above)

my @fib = 1, 1, ** ... *; # lazy infinite list of fibonacci series,
# computed using a closure!
my @fib = 1, 1, -> $a, $b { $a + $b } ... *; # (equivalent to the above)
my @fib = 1, 1, { $^a + $^b } ... *; #(... also equivalent to the above)
# $a and $b will always take the previous values, meaning here
# they'll start with $a = 1 and $b = 1 (values we set by hand).
# then $a = 1 and $b = 2 (result from previous $a+$b), and so on.

say @fib[~10]; #=> 1 1 2 3 5 8 13 21 34 55
# (using a range as the index)
# Note : as for ranges, once reified, elements aren't re-calculated.
# That's why `@primes[~100]` will take a long time the first time you print
# it, then be instant.

### Regular Expressions
# I'm sure a lot of you have been waiting for this one.
# Well, now that you know a good deal of Perl 6 already, we can get started.
# First off, you'll have to forget about "PCRE regexps" (perl-compatible regexps).
#
# IMPORTANT: Don't skip them because you know PCRE. They're different.
# Some things are the same (like `?`, `+`, and `*`),
# but sometimes the semantics change (`|`).
# Make sure you read carefully, because you might trip over a new behavior.
#
# Perl 6 has many features related to RegExps. After all, Rakudo parses itself.
# We're first going to look at the syntax itself,
# then talk about grammars (PEG-like), differences between

```

```

# `token`, `regex` and `rule` declarators, and some more.
# Side note: you still have access to PCRE regexps using the `:P5` modifier.
# (we won't be discussing this in this tutorial, however)
#
# In essence, Perl 6 natively implements PEG ("Parsing Expression Grammars").
# The pecking order for ambiguous parses is determined by a multi-level
# tie-breaking test:
# - Longest token matching. `foo\s+` beats `foo` (by 2 or more positions)
# - Longest literal prefix. `food\w*` beats `foo\w*` (by 1)
# - Declaration from most-derived to less derived grammars
#   (grammars are actually classes)
# - Earliest declaration wins
say so 'a' ~~ /a/; #=> True
say so 'a' ~~ / a /; # More readable with some spaces!

# In all our examples, we're going to use the smart-matching operator against
# a regexp. We're converting the result using `so`, but in fact, it's
# returning a `Match` object. They know how to respond to list indexing,
# hash indexing, and return the matched string.
# The results of the match are available as `$/` (implicitly lexically-scoped).
# You can also use the capture variables (`$0`, `$1`, ... starting at 0, not 1 !).
#
# You can also note that `~~` does not perform start/end checking
# (meaning the regexp can be matched with just one char of the string),
# we're going to explain later how you can do it.

# In Perl 6, you can have any alphanumeric as a literal,
# everything else has to be escaped, using a backslash or quotes.
say so 'a|b' ~~ / a '|' b /; # `True`. Wouldn't mean the same if `|` wasn't escaped
say so 'a|b' ~~ / a \| b /; # `True`. Another way to escape it.

# The whitespace in a regexp is actually not significant,
# unless you use the `:s` (':sigspace', significant space) modifier.
say so 'a b c' ~~ / a b c /; # `False`. Space is not significant here
say so 'a b c' ~~ /:s a b c /; # `True`. We added the modifier `:s` here.

# It is, however, important as for how modifiers (that you're gonna see just below)
# are applied ...

## Quantifying - `?`, `+`, `*` and `**`.
# - `?` - 0 or 1
so 'ac' ~~ / a b c /; # `False`
so 'ac' ~~ / a b? c /; # `True`, the "b" matched 0 times.
so 'abc' ~~ / a b? c /; # `True`, the "b" matched 1 time.

# ... As you read just before, whitespace is important because it determines
# which part of the regexp is the target of the modifier:
so 'def' ~~ / a b c? /; # `False`. Only the `c` is optional
so 'def' ~~ / ab?c /; # `False`. Whitespace is not significant
so 'def' ~~ / 'abc'? /; # `True`. The whole "abc" group is optional.

# Here (and below) the quantifier applies only to the `b`

# - `+` - 1 or more

```

```

so 'ac' ~~ / a b+ c /; # `False`; `+` wants at least one matching
so 'abc' ~~ / a b+ c /; # `True`; one is enough
so 'abbbbc' ~~ / a b+ c /; # `True`, matched 4 "b"s

# - `*` - 0 or more
so 'ac' ~~ / a b* c /; # `True`, they're all optional.
so 'abc' ~~ / a b* c /; # `True`
so 'abbbbc' ~~ / a b* c /; # `True`
so 'aec' ~~ / a b* c /; # `False`. "b"(s) are optional, not replaceable.

# - `**` - (Unbound) Quantifier
# If you squint hard enough, you might understand
# why exponentiation is used for quantity.
so 'abc' ~~ / a b ** 1 c /; # `True` (exactly one time)
so 'abc' ~~ / a b ** 1..3 c /; # `True` (one to three times)
so 'abbbc' ~~ / a b ** 1..3 c /; # `True`
so 'abbbbbbc' ~~ / a b ** 1..3 c /; # `False` (too much)
so 'abbbbbbc' ~~ / a b ** 3..* c /; # `True` (infinite ranges are okay)

# - `[<[]>` - Character classes
# Character classes are the equivalent of PCRE's `[]` classes, but
# they use a more perl6-ish syntax:
say 'fooa' ~~ / f [<[ o a ]>+ /; #=> 'fooa'
# You can use ranges:
say 'aeiou' ~~ / a [<[ e..w ]> /; #=> 'ae'
# Just like in normal regexes, if you want to use a special character, escape it
# (the last one is escaping a space)
say 'he-he !' ~~ / 'he-' [<[ a..z \! \ ]> + /; #=> 'he-he !'
# You'll get a warning if you put duplicate names
# (which has the nice effect of catching the wrote quoting:)
'he he' ~~ / [<[ h e ' ' ]> /; # Warns "Repeated characters found in characters class"

# You can also negate them ... (equivalent to `[~` in PCRE)
so 'foo' ~~ / <-[ f o ]> + /; # False

# ... and compose them: :
so 'foo' ~~ / [<[ a..z ] - [ f o ]> + /; # False (any letter except f and o)
so 'foo' ~~ / <-[ a..z ] + [ f o ]> + /; # True (no letter except f and o)
so 'foo!' ~~ / <-[ a..z ] + [ f o ]> + /; # True (the + doesn't replace the left part)

## Grouping and capturing
# Group: you can group parts of your regexp with `[]`.
# These groups are not captured (like PCRE's `(?:)`).
so 'abc' ~~ / a [ b ] c /; # `True`. The grouping does pretty much nothing
so 'foo012012bar' ~~ / foo [ '01' [<[0..9]> ] + bar /;
# The previous line returns `True`.
# We match the "012" 1 or more time (the `+` was applied to the group).

# But this does not go far enough, because we can't actually get back what
# we matched.
# Capture: We can actually capture the results of the regexp, using parentheses.
so 'fooABCABcbar' ~~ / foo ( 'A' [<[A..Z]> 'C' ) + bar /; # `True`. (using `so` here, `$/` below)

# So, starting with the grouping explanations.

```

```

# As we said before, our `Match` object is available as `$/`:
say $/; # Will print some weird stuff (we'll explain) (or "Nil" if nothing matched).

# As we also said before, it has array indexing:
say $/[0]; #=> ABC ABC
      # These weird brackets are `Match` objects.
      # Here, we have an array of these.
say $0; # The same as above.

# Our capture is `$0` because it's the first and only one capture in the regexp.
# You might be wondering why it's an array, and the answer is simple:
# Some capture (indexed using `$0`, `$/[0]` or a named one) will be an array
# IFF it can have more than one element
# (so, with `*`, `+` and `**` (whatever the operands), but not with `?`).
# Let's use examples to see that:
so 'fooABCbar' ~~ / foo ( A B C )? bar /; # `True`
say $/[0]; #=> ABC
say $0.WHAT; #=> (Match)
      # It can't be more than one, so it's only a single match object.
so 'foobar' ~~ / foo ( A B C )? bar /; #=> True
say $0.WHAT; #=> (Any)
      # This capture did not match, so it's empty
so 'foobar' ~~ / foo ( A B C ) ** 0..1 bar /; # `True`
say $0.WHAT; #=> (Array)
      # A specific quantifier will always capture an Array,
      # may it be a range or a specific value (even 1).

# The captures are indexed per nesting. This means a group in a group will be nested
# under its parent group: `$/[0][0]`, for this code:
'hello--world' ~~ / ( 'hello' ( <[ \- \~ ]> + ) ) 'world' /;
say $/[0].Str; #=> hello~
say $/[0][0].Str; #=> ~

# This stems from a very simple fact: `$/` does not contain strings, integers or arrays,
# it only contains match objects. These contain the `.list`, `.hash` and `.Str` methods.
# (but you can also just use `match<key>` for hash access
# and `match[idx]` for array access)
say $/[0].list.perl; #=> (Match.new(...)).list
      # We can see it's a list of Match objects. Those contain
      # a bunch of infos: where the match started/ended,
      # the "ast" (see actions later), etc.
      # You'll see named capture below with grammars.

## Alternatives - the `or` of regexps
# WARNING: They are DIFFERENT from PCRE regexps.
so 'abc' ~~ / a [ b | y ] c /; # `True`. Either "b" or "y".
so 'ayc' ~~ / a [ b | y ] c /; # `True`. Obviously enough ...

# The difference between this `|` and the one you're used to is LTM.
# LTM means "Longest Token Matching". This means that the engine will always
# try to match as much as possible in the string
'foo' ~~ / fo | foo /; # `foo`, because it's longer.
# To decide which part is the "longest", it first splits the regex in two parts:
# The "declarative prefix" (the part that can be statically analyzed)

```



```

# and the procedural parts.
# Declarative prefixes include alternations (`|`), conjunctions (`&`),
# sub-rule calls (not yet introduced), literals, characters classes and quantifiers.
# The latter include everything else: back-references, code assertions,
# and other things that can't traditionnaly be represented by normal regexps.
#
# Then, all the alternatives are tried at once, and the longest wins.
# Examples:
# DECLARATIVE / PROCEDURAL
/ 'foo' \d+      [ <subrule1> || <subrule2> ] /;
# DECLARATIVE (nested groups are not a problem)
/ \s* [ \w & b ] [ c | d ] /;
# However, closures and recursion (of named regexps) are procedural.
# ... There are also more complicated rules, like specificity
# (literals win over character classes)

# Note: the first-matching `or` still exists, but is now spelled `||`
'foo' ~~ / fo || foo /; # `fo` now.

```

```

### Extra: the MAIN subroutine
# The `MAIN` subroutine is called when you run a Perl 6 file directly.
# It's very powerful, because Perl 6 actually parses the arguments
# and pass them as such to the sub. It also handles named argument (`--foo`)
# and will even go as far as to autogenerate a `--help`
sub MAIN($name) { say "Hello, $name !" }
# This produces:
#   $ perl6 cli.pl
#   Usage:
#   t.pl <name>

# And since it's a regular Perl 6 sub, you can haz multi-dispatch:
# (using a "Bool" for the named argument so that we can do `--replace`
# instead of `--replace=1`)
subset File of Str where *.IO.d; # convert to IO object to check the file exists

multi MAIN('add', $key, $value, Bool :$replace) { ... }
multi MAIN('remove', $key) { ... }
multi MAIN('import', File, Str :$as) { ... } # omitting parameter name
# This produces:
#   $ perl6 cli.pl
#   Usage:
#   t.pl [--replace] add <key> <value>
#   t.pl remove <key>
#   t.pl [--as=<Str>] import (File)
# As you can see, this is *very* powerful.
# It even went as far as to show inline the constants.
# (the type is only displayed if the argument is `$`/is named)

###
### APPENDIX A:
###

```

```

### List of things
###

# It's considered by now you know the Perl6 basics.
# This section is just here to list some common operations,
# but which are not in the "main part" of the tutorial to bloat it up

## Operators

## * Sort comparison
# They return one value of the `Order` enum : `Less`, `Same` and `More`
# (which numerify to -1, 0 or +1).
1 <=> 4; # sort comparison for numerics
'a' leg 'b'; # sort comparison for string
$obj eqv $obj2; # sort comparison using eqv semantics

## * Generic ordering
3 before 4; # True
'b' after 'a'; # True

## * Short-circuit default operator
# Like `or` and `||`, but instead returns the first *defined* value :
say Any // Nil // 0 // 5; #=> 0

## * Short-circuit exclusive or (XOR)
# Returns `True` if one (and only one) of its arguments is true
say True ^^ False; #=> True

## * Flip Flop
# The flip flop operators (`ff` and `fff`, equivalent to P5's `..`/`...`).
# are operators that take two predicates to test:
# They are `False` until their left side returns `True`, then are `True` until
# their right side returns `True`.
# Like for ranges, you can exclude the iteration when it became `True`/`False`
# by using `^^` on either side.
# Let's start with an example :
for <well met young hero we shall meet later> {
    # by default, `ff`/`fff` smart-match (`~~`) against `$_`:
    if 'met' ~ff 'meet' { # Won't enter the if for "met"
        # (explained in details below).

        .say
    }

    if rand == 0 ff rand == 1 { # compare variables other than `$_`
        say "This ... probably will never run ...";
    }
}

# This will print "young hero we shall meet" (excluding "met"):
# the flip-flop will start returning `True` when it first encounters "met"
# (but will still return `False` for "met" itself, due to the leading `^^`
# on `ff`), until it sees "meet", which is when it'll start returning `False`.

# The difference between `ff` (awk-style) and `fff` (sed-style) is that
# `ff` will test its right side right when its left side changes to `True`,

```

```

# and can get back to `False` right away
# (*except* it'll be `True` for the iteration that matched) -
# While `fff` will wait for the next iteration to
# try its right side, once its left side changed:
.say if 'B' ff 'B' for <A B C B A>; #=> B B
# because the right-hand-side was tested
# directly (and returned `True`).
# "B"s are printed since it matched that time
# (it just went back to `False` right away).

.say if 'B' fff 'B' for <A B C B A>; #=> B C B
# The right-hand-side wasn't tested until
# `$_` became "C"
# (and thus did not match instantly).

# A flip-flop can change state as many times as needed:
for <test start print it stop not printing start print again stop not anymore> {
    .say if $_ eq 'start' ^ff^ $_ eq 'stop'; # exclude both "start" and "stop",
#=> "print it print again"
}

# you might also use a Whatever Star,
# which is equivalent to `True` for the left side or `False` for the right:
for (1, 3, 60, 3, 40, 60) { # Note: the parenthesis are superfluous here
    # (sometimes called "superstitious parentheses")
    .say if $_ > 50 ff *; # Once the flip-flop reaches a number greater than 50,
# it'll never go back to `False`
#=> 60 3 40 60
}

# You can also use this property to create an `If`
# that'll not go through the first time :
for <a b c> {
    .say if * ^ff *; # the flip-flop is `True` and never goes back to `False`,
# but the `^` makes it *not run* on the first iteration
#=> b c
}

# - `===` is value identity and uses `.WHICH` on the objects to compare them
# - `:=` is container identity and uses `VAR()` on the objects to compare them

```

If you want to go further, you can:

- Read the Perl 6 Advent Calendar. This is probably the greatest source of Perl 6 information, snippets and such.
- Come along on [#perl6](#) at [irc.freenode.net](#). The folks here are always helpful.
- Check the source of Perl 6's functions and classes. Rakudo is mainly written in Perl 6 (with a lot of NQP, “Not Quite Perl”, a Perl 6 subset easier to implement and optimize).
- Read the language design documents. They explain P6 from an implementor point-of-view, but it's still very interesting.