Perl 6 Introduction

This document is intended to give you a quick overview of the Perl 6 programming language.
For those who are new to Perl 6 it should get you up and running.

Some sections of this document reference other (more complete and accurate) parts of the Perl 6 documentation. You should read them if you need more information on a specific subject.

Throughout this document, you will find examples for most discussed topics. To better understand them, take the time to reproduce all examples.

License

Contribution
If you would like to contribute to this document head over to:
http://github.com/hankache/perl6intro

1. Introduction

1.1. What is Perl 6

Perl 6 is a high-level, general-purpose, gradually typed language. Perl 6 is multi-paradigmatic. It supports Procedural, Object Oriented and Functional programming.

Perl 6 motto:

- TMTOWTDI (Pronounced Tim Toady): There is more than one way to do it.
- Easy things should stay easy, hard things should get easier, and impossible things should get hard.

1.2. Jargon

Perl 6: Is a language specification with a test suite. Implementations that pass the specification test suite are considered Perl 6.
Rakudo: Is a compiler for Perl 6.
Rakudobrew: Is a Perl 6 installation manager.
Panda: Is a Perl 6 module installer.

1.3. Installing Perl 6

Linux

1. Install Rakudobrew: https://github.com/tadzik/rakudobrew
2. Install Rakudo: Type the following command in the terminal `rakudobrew build moar`
3. Install Panda: Type the following command in the terminal `rakudobrew build-panda`

OSX
Follow the same steps listed for installing on Linux
OR
Install with homebrew: `brew install rakudo-star`

Windows

1. Download the latest installer (file with .MSI extension) from http://rakudo.org/downloads/star/
   If your system architecture is 32-bit, download the x86 file if it's 64-bit download the x86_64 file.
2. After installation add C:\rakudo\bin to your PATH

Docker

1. Get the official Docker image `docker pull rakudo-star`
2. Then run a container with the image `docker run -it rakudo-star`

1.4. Running Perl 6 code

Running Perl 6 code can be done using the REPL (Read-Eval-Print Loop).
Within the terminal, type `perl6` , write your code and hit [Enter]

Alternatively, write your code in a file, save it and run it.
It is recommended that Perl 6 files have a `.pl6` extension.

Run the file from the terminal using the following syntax: `perl6 filename.pl6`

The REPL is mostly used for trying a specific piece of code, typically a single line.
For programs with more than a single line it is recommended to store them in a file and then run them.

1.5. Editors

Since most of the time we will be writing and storing our Perl 6 programs in files, we should have a decent text editor that recognizes Perl 6 syntax.

I personally use and recommend Atom. It is a modern text editor and comes with Perl 6 syntax highlighting out of the box.

Other people in the community also use Vim, Emacs or Padre.

Recent versions of Vim ship with syntax highlighting out of the box. Emacs and Padre will require installation of additional packages.

1.6. Hello World!

We shall begin with The `hello world` ritual.

```
say 'hello world';
```

1.7. Syntax overview

Perl 6 is **free form**: You are free (most of the time) to use any amount of whitespace.

Statements are typically a logical line of code, they need to end with a semicolon: `if True { say "Hello" };`

Expressions are a special type of statement that returns a value: `1+2` will return `3`

Expressions are made of **Terms** and **Operators**.

Terms are:

- **Variables**: A value that can be manipulated and changed.
- **Literals**: A constant value like a number or a string.

Operators are classified into types:

Type	Explanation	Example
Prefix	Before the term.	`++1`
Infix	Between terms	`1+2`
Postfix	After the term	`1++`
Circumfix	Around the term	`(1)`
Postcircumfix	After one term, around another	`Array[1]`

1.7.1. Identifiers

Identifiers are the name given to terms when you define them.

Rules:

- They must start with an alphabetic character or an underscore.
- They can contain digits (except the first character).
- They can contain dashes or apostrophes (except the first and last character), provided there's an alphabetic character to the right side of each dash/apostrophe.

Valid	**Invalid**
`var1`	`1var`
`var-one`	`var-1`
`var'one`	`var'1`
`var1_`	`var1'`
`_var`	`-var`

Naming conventions:

- Camel case: `variableNo1`
- Kebab case: `variable-no1`
- Snake case: `variable_no1`

You are free to name your identifiers as you like, but it is good practice to adopt one naming convention consistently.

Using meaningful names will ease your (and others) programming life.
`var1 = var2 * var3` is syntactically correct but its purpose is not evident.
`monthly-salary = daily-rate * working-days` would be a better way to name your variables.

1.7.2. Comments

A comment is a piece of text ignored by the compiler and used as a note.

Comments are divided into 3 types:

- Single line:

```
#This is a single line comment
```

- Embedded:

```
say #`(This is an embedded comment) "Hello World."
```

- Multi line:

```
=begin comment
This is a multi line comment.
Comment 1
Comment 2
=end comment
```

1.7.3. Quotes

Strings need to be delimited by either double quotes or single quotes.

Always use double quotes:

- if your string contains an apostrophe.
- if your string contains a variable that needs to be interpolated.

```
say 'Hello World';   #Hello World
say "Hello World";   #Hello World
say "Don't";         #Don't
my $name = 'John Doe';
say 'Hello $name';   #Hello $name
say "Hello $name";   #Hello John Doe
```

2. Operators

Operator	Type	Description	Example	Result
`+`	`Infix`	Addition	`1 + 2`	`3`
`-`	`Infix`	Subtraction	`3 - 1`	`2`
`*`	`Infix`	Multiplication	`3 * 2`	`6`
`**`	`Infix`	Power	`3 ** 2`	`9`
`/`	`Infix`	Division	`3 / 2`	`1.5`

Note	For the complete list of operators, including their precedence, go to http://doc.perl6.org/language/operators

3. Variables

Perl 6 variables are classified into 3 categories: Scalars, Arrays and Hashes.

A **sigil** (Sign in Latin) is a character that is used as a prefix to categorize variables.

- `$` is used for scalars
- `@` is used for arrays
- `%` is used for hashes

3.1. Scalars

A scalar holds one value or reference.

```
#String
my $name = 'John Doe';
say $name;

#Integer
my $age = 99;
say $age;
```

3.2. Arrays

Arrays are lists containing multiple values.

```
my @animals = ['camel','llama','owl'];
say @animals;
```

Many operations can be done on arrays as shown in the below example:

Tip	The tilde ~ is used for concatenation.

Script

```
my @animals = ['camel','vicuña','llama'];
say "The zoo contains " ~ @animals.elems ~ " animals";
say "The animals are: " ~ @animals;
say "I will adopt an owl for the zoo";
@animals.push("owl");
say "Now my zoo has: " ~ @animals;
say "The first animal we adopted was the " ~ @animals[0];
@animals.pop;
say "Unfortunately the owl got away and we're left with: " ~ @animals;
say "We're closing the zoo and keeping one animal only";
say "We're going to let go: " ~ @animals.splice(1,2) ~ " and keep the " ~ @animals;
```

Output

```
The zoo contains 3 animals
The animals are: camel vicuña llama
I will adopt an owl for the zoo
Now my zoo has: camel vicuña llama owl
The first animal we adopted was the camel
Unfortunately the owl got away and we're left with: camel vicuña llama
We're closing the zoo and keeping one animal only
We're going to let go: vicuña llama and keep the camel
```

Explanation

`.elems` returns the number of elements in an array.

`.push()` adds an element to the array.

We can access a specific element in the array by specifying its position `@animal[0]`.

`.pop` removes the last element from the array.

`.splice(a,b)` will remove the `b` elements that start at position `a`.

Note	For the complete Array reference, see http://doc.perl6.org/type/Array

3.3. Hashes

A Hash is a set of Key/Value pairs.

```
my %capitals = ('London','UK','Berlin','Germany');
say %capitals;

#another succinct way of filling the hash:
my %capitals = (London => 'UK', Berlin => 'Germany');
say %capitals;
```

3.4. Types

In the previous examples, we did not specify what type the variables should hold. This means that the variable type will be whatever we assigned to it.

Tip	`.WHAT` will return the type of the variable.

```
my $var = 'Text';
say $var;
say $var.WHAT;

$var = 123;
say $var;
say $var.WHAT;
```

As you can see in the above example, the type of `$var` was once (Str) and then (Int).

This style of programming is called dynamic typing. Dynamic in the sense that a variable type is whatever you assign to it.

Now try running the below example:
Notice `Str` before the variable name.

```
my Str $var = 'Text';
say $var;
say $var.WHAT;

$var = 123;
say $var;
say $var.WHAT;
```

It will fail and return this error message: `Type check failed in assignment to $var; expected Str but got Int`

What happened is that we specified beforehand that the variable should be of type (Str). When we tried to assign an (Int) to it, it failed.

This style of programming is called static typing. Static in the sense that variable types are defined before assignment

and cannot change.

Perl 6 is classified as **gradually typed**; it allows both **static** and **dynamic** typing.

Below is a list of the most commonly used types.
You will most probably never use the first two but they are listed for informational purpose.

Type	**Description**	**Example**	**Result**
Mu	The root of the Perl 6 type hierarchy		
Any	Default base class for new classes and for most built-in classes		
Cool	Value that can be treated as a string or number interchangeably	my Cool $var = 31; say $var.flip; say $var * 2;	13 62
Str	String of characters	my Str $var = "NEON"; say $var.flip;	NOEN
Int	Integer (arbitrary-precision)	7 + 7	14
Rat	Rational number (limited-precision)	0.1 + 0.2	0.3
Bool	Boolean	!True	False

3.5. Introspection

Introspection is the process of getting information about an object properties like its type.
In one of the previous example we used `.WHAT` to return the type of the variable.

```
my Int $var;
say $var.WHAT;
my $var2;
say $var2.WHAT;
$var2 = 1;
say $var2.WHAT;
$var2 = "Hello";
say $var2.WHAT;
$var2 = True;
say $var2.WHAT;
$var2 = Nil;
say $var2.WHAT;
```

```
(Int)
(Any)
(Int)
(Str)
(Bool)
(Any)
```

The type of a variable holding a value is correlated to its value.
The type of a strongly declared empty variable is the type with which it was declared.
The type of an empty variable that wasn't strongly declared is `(Any)`
To clear the value of a variable, assign `Nil` to it.

3.6. Scoping

Before using a variable for the first time, it needs to be declared.

Several declarators are used in Perl 6, `my` is what we have been using so far in the examples above.

```
my $var=1;
```

The `my` declarator give the variable **lexical** scope. In other words, the variable will only be accessible in the same block it was declared.

A block in Perl 6 is delimited by `{ }` . If no block is found, the variable will be available in the whole Perl script.

```
{
  my Str $var = 'Text';
  say $var; #is accessible
}
say $var; #is not accessible, returns an error
```

Since a variable is only accessible in the block where it is defined, the same variable name can be redefined in another block.

```
{
  my Str $var = 'Text';
  say $var;
}
my Int $var = 123;
say $var;
```

Note	For more info on variables, see http://doc.perl6.org/language/variables

4. Functions and mutators

It is important to differentiate between functions and mutators.
Functions do no change the initial state of the object they were called on.
Mutators modify the state of the object.

```
Script

  my @numbers = [7,2,4,9,11,3];

  @numbers.push(99);
  say @numbers;       #1

  say @numbers.sort; #2
  say @numbers;       #3

  @numbers.=sort;
  say @numbers;       #4


Output

  [7 2 4 9 11 3 99] #1
  (2 3 4 7 9 11 99) #2
  [7 2 4 9 11 3 99] #3
  [2 3 4 7 9 11 99] #4
```

Explanation

`.push` is a mutator, it changes the state of the array (#1)

`.sort` is a function, it returns a sorted array but doesn't modify the state of the initial array:

- (#2) shows that it returned a sorted array.
- (#3) shows that the initial array is still unmodified.

In order to enforce a function to act as a mutator, we use `.=` instead of `.` (#4) (Line 9 of the script)

5. Loops and conditions

Perl 6 has a multitude of conditionals and looping constructs.

5.1. if

The code runs only if the condition has been met.

```
my $age = 19;

if $age > 18 {
  say 'Welcome'
};
```

In Perl 6 we can invert the code and the condition.
Even if the code and the condition have been inverted, the condition is always evaluated first.

```
my $age = 19;

say 'Welcome' if $age > 18;
```

If the condition is not met, we can still specify alternative blocks for execution using:

- `else`
- `elsif`

```
#run the same code for different values of the variable
my $number-of-seats = 9;

if $number-of-seats <= 5 {
  say 'I am a sedan'
} elsif $number-of-seats <= 7 {
  say 'I am 7 seater'
} else {
  say 'I am a van'
};
```

5.2. unless

The negated version of an if statement can be written using `unless` .

The following code:

```
my $clean-shoes = False;

if  not $clean-shoes {
  say 'Clean your shoes'
};
```

can be written as:

```
my $clean-shoes = False;

unless $clean-shoes {
  say 'Clean your shoes'
};
```

Negation in Perl 6 is done using either `!` or `not`.

`unless (condition)` is used instead of `if not (condition)`.

`unless` cannot have an `else` clause.

5.3. with

`with` behave like the `if` statement, but checks if the variable is defined.

```
my Int $var=1;

with $var {
  say 'Hello'
};
```

If you run the code without assigning a value to the variable nothing should happen.

```
my Int $var;

with $var {
  say 'Hello'
};
```

`without` is the negated version of `with`. You should be able to relate it to `unless`.

If the first `with` condition is not met, an alternate path can be specified using `orwith`.
`with` and `orwith` can be compared to `if` and `elsif`.

5.4. for

The `for` loop iterates over multiple values.

```
my @array = [1,2,3];

for @array -> $array-item {
  say $array-item*100
};
```

Notice that we created an iteration variable `$array-item` in order to perform the operation `*100` on each array item.

5.5. given

`given` is the Perl 6 equivalent of the switch statement in other languages.

```
my $var = 42;

given $var {
    when 0..50 { say 'Less than 50'}
    when Int { say "is an Int" }
    when 42  { say 42 }
    default  { say "huh?" }
};
```

After a successful match, the matching process will stop.

Alternatively `proceed` will instruct Perl 6 to continue matching even after a successful match.

```
my $var = 42;

given $var {
    when 0..50 { say 'Less than 50';proceed}
    when Int { say "is an Int";proceed}
    when 42  { say 42 }
    default  { say "huh?" }
};
```

5.6. loop

`loop` is another way of writing a `for` loop.

Actually `loop` is how `for` loops are written in C-family programming languages.

Perl 6 belongs to the C-family languages.

```
loop (my $i=0; $i < 5; $i++) {
  say "The current number is $i"
};
```

Note	For more info on loops and conditions, see http://doc.perl6.org/language/control

6. I/O

In Perl 6, two of the most common *Input/Output* interfaces are the *Terminal* and *Files*.

6.1. Basic I/O using the Terminal

6.1.1. say

`say` writes to the standard output. It appends a new line at the end. In other words, the following code:

```
say 'Hello Mam.';
say 'Hello Sir.';
```

will be written on 2 separate lines.

6.1.2. print

`print` on the other hand behave like `say` but doesn't add a new line.

Try replacing `say` with `print` and compare both results.

6.1.3. get

`get` is used to capture input from the terminal.

```
my $name;

say "Hi, what's your name?";
$name=get;

say "Dear $name welcome to Perl 6";
```

When the above code runs, the terminal will be waiting for you to input your name. Subsequently, it will greet you.

6.1.4. prompt

`prompt` is a combination of `print` ans `get` .

The above example can be written like this:

```
my $name = prompt("Hi, what's your name? ");

say "Dear $name welcome to Perl 6";
```

6.2. Running Shell Commands

Two subroutines can be used to run shell commands:

- `run` Runs an external command without involving a shell
- `shell` Runs a command through the system shell. All shell meta characters are interpreted by the shell, including pipes, redirects, environment variable substitutions and so on

```
my $name = 'Neo';
my $command = run 'echo', "hello $name";
my $command2 = shell "ls";
```

`echo` and `ls` are common shell keywords.
`echo` prints text to the terminal (the equivalent of `print` in Perl 6)
`ls` lists all files and folders in the current directory

6.3. File I/O

6.3.1. slurp

`slurp` is used to read data from a file.

Create a text file with the following content:

datafile.txt

```
John 9
Johnnie 7
Jane 8
Joanna 7
```

```
my $data = slurp "datafile.txt";
say $data;
```

6.3.2. spurt

`spurt` is used to write data to a file.

```
my $newdata = "New scores:
Paul 10
Paulie 9
Paulo 11";

spurt "newdatafile.txt", $newdata;
```

After running the above code, a new file named *newdatafile.txt* will be created. It will contain the new scores.

6.4. Working with files and directories

Perl 6 can list the contents of a directory without running shell commands (using `ls`) as seen in a previous example.

```
say dir;              #List files and folders in the current directory
say dir "/Documents"; #List files and folders in the specified directory
```

In addition to that you can create new directories and delete them.

```
mkdir "newfolder";
rmdir "newfolder";
```

`mkdir` creates a new directory.
`rmdir` delete an empty directory. Returns an error if not empty.

You can also check if the specified path exits, if it is a file or a directory:

In the directory where you will be running the below script, create an empty folder `folder123` and an empty pl6 file `script123.pl6`

```
say "script123.pl6".IO.e;
say "folder123".IO.e;

say "script123.pl6".IO.d;
say "folder123".IO.d;

say "script123.pl6".IO.f;
say "folder123".IO.f;
```

`IO.e` checks if the directory/file exist.
`IO.f` checks if the path is a file.
`IO.d` checks if the path is a directory.

Note	For more info on I/O, see http://doc.perl6.org/type/IO

7. Subroutines

7.1. Definition

Subroutines or **subs** are a means of packaging a set of functionality.

A subroutine definition begins with the keyword `sub` . After their definition, they can be called by their handle.
Check out the below example:

```
sub alien-greeting {
  say "Hello earthlings";
}

alien-greeting;
```

The previous example showcased a subroutine that doesn't require any input.

7.2. Signature

Many subroutines would require some input in order to work. That input is provided by **arguments**. The number and type of arguments that this subroutine accepts is called its **signature**.

The below subroutine accepts a string argument.

```
sub say-hello (Str $name) {
    say "Hello " ~ $name ~ "!!!!"
}
say-hello "Paul";
say-hello "Paula";
```

7.3. Multi subroutine

It is possible to define multiple subroutines having the same name but different signatures. When the subroutine is called, the runtime environment will decide which version to use depending on the number and type of the supplied arguments. This type of subroutines is defined the same way as normal subs with the exception of swapping the `sub` keyword with `multi` .

```
multi greet($name) {
    say "Good morning $name";
}
multi greet($name, $title) {
    say "Good morning $title $name";
}

greet "Johnnie";
greet "Laura","Mrs.";
```

7.4. Default and Optional Arguments

If a subroutine is defined to accept an argument, and we call it without providing it with the required argument, it will fail.

Alternatively Perl 6 provides us the ability to define subroutines with:

- Optional Arguments
- Default Arguments

Optional arguments are defined by appending `?` after the argument name.

```
sub say-hello($name?) {
  if $name.defined {
    say "Hello " ~ $name;
  } else {
    say "Hello Human";
  }
}
say-hello;
say-hello("Laura");
```

If the user doesn't supply an argument, it can default to a specific value.
This is done by assigning a value to the argument within the subroutine definition.

```
sub say-hello($name="Matt") {
  say "Hello " ~ $name;
}
say-hello;
say-hello("Laura");
```

Note	For more info on subroutines and functions, see http://doc.perl6.org/language/functions

8. Classes & Objects

8.1. Introduction

Object Oriented programming is one of the widely used paradigms nowadays.
An **object** is a set of variables and subroutines bundled together.
The variables are called **attributes** and the subroutines are called **methods**.
Attributes define the **state** and methods define the **behavior** of an object.

A **class** defines the structure of a set of **objects**.

In order to understand the relationship consider the below example:

There are 4 people present in a room	**objects** ⇒ 4 people
These 4 people are humans	**class** ⇒ Human
They have different names, age, sex and nationality	**attributes** ⇒ name, age, sex, nationality

In *object oriented* parlance, we say that objects are **instances** of a class.

Consider the below script:

```
class Human {
has $name;
has $age;
has $sex;
```

```
has $nationality;
}

my $john = Human.new(name => 'John', age => 23, sex => 'M', nationality => 'American');
say $john;
```

The `class` keyword is used to define a class.
The `has` keyword is used to define attributes of a class.
The `.new()` method is called a **constructor**. It creates the object as an instance of the class it has been called on.

In the above script, a new variable `$john` holds a reference to a new instance of "Human" defined by `Human.new()`.
The arguments passed to the `.new()` method are used to set the attributes of the underlying object.

A class can be given *lexical scope* using `my`:

```
my class Human {

}
```

8.2. Encapsulation

Encapsulation is an object oriented concept that bundles a set of data and methods together.
The data (attributes) within an object should be **private**, in other words, accessible only from within the object.
In order to access the attributes from outside the object we use methods that we call **accessors**.

The below two scripts have the same result.

Direct access to the variable:

```
my $var = 7;
```

```
say $var;
```

Encapsulation:

```
my $var = 7;
sub sayvar {
  $var;
}
say sayvar;
```

The method `sayvar` is an accessors. It let us access the value of the variable without getting direct access to it.

Encapsulation is facilitated in Perl 6 with the use of **twigils**.
Twigils are secondary *sigils*. They come between the sigil and the attribute name.
Two twigils are used in classes:

- `!` is used to explicitly declare that the attribute is private.
- `.` is used to automatically generate an accessor for the attribute.

By default, all attributes are private but it is a good habit to always use the `!` twigil.

In line with what we said we should rewrite the above class as following:

```
class Human {
has $!name;
has $!age;
has $!sex;
has $!nationality;
}
```

```
my $john = Human.new(name => 'John', age => 23, sex => 'M', nationality => 'American');
say $john;
```

Append to the script the following statement: `say $john.age;`
It will return the following error: `Method 'age' not found for invocant of class 'Human'`
The reason being that `$!age` is private and can only be used within the object. Trying to access it outside the object will return an error.

Now replace `has $!age` with `has $.age` and see what will be the result of `say $john.age;`

8.3. Named vs. Positional Arguments

In Perl 6, all classes inherit a default `.new()` constructor.
It can be used to create objects by providing it with arguments.
The default constructor can only be provided with **named arguments**.
If you consider the above example, you'll remark that all the arguments supplied to `.new()` are defined by name:

- name ⇒ 'John'
- age ⇒ 23

What if i do not want to supply the name of each attribute each time i want to create a new object?
Then I need to create another constructor that accepts **positional arguments**.

```
class Human {
  has $.name;
  has $.age;
  has $.sex;
  has $.nationality;
```

```
  #new constructor that overrides the default one.
  method new ($name,$age,$sex,$nationality) {
    self.bless(:$name,:$age,:$sex,:$nationality);
  }
}

my $john = Human.new('John',23,'M','American');
say $john;
```

The constructor that accepts positional arguments need to be defined as seen above.

8.4. Methods

8.4.1. Introduction

Methods are the *subroutines* of an object.
Like subroutines, they are a means of packaging a set of functionality, they accept **arguments**, have a **signature** and can be defined as **multi**.

Methods are defined using the `method` keyword.
In normal circumstances, methods a required to perform some sort of action on the objects' attributes. This enforces the concept of encapsulation. Object attributes can only be manipulated from within the object using methods. The outside world, can only interact with the object methods, and has no access to its attributes.

```
class Human {
  has $.name;
  has $.age;
  has $.sex;
  has $.nationality;
  has $.eligible;
```

```
    method assess-eligibility {
        if self.age < 21 {
          $!eligible = 'No'
        } else {
          $!eligible = 'Yes'
        }
    }

}

my $john = Human.new(name => 'John', age => 23, sex => 'M', nationality => 'American');
$john.assess-eligibility;
say $john.eligible;
```

Once methods are defined within a class, they can be called on an object using the *dot notation*:
object **.** *method* or as in the above example: `$john.assess-eligibility`

Within the definition of a method, if we need to reference the object itself to call another method we use the `self` keyword.

Within the definition of a method, if we need to reference an attribute we use `!` even if it was defined with `.`
The rationale being that what the `.` twigil does is declare an attribute with `!` and automate the creation of an accessor.

In the above example `if self.age < 21` and `if $!age < 21` would have the same effect, although they are technically different:

- `self.age` calls the `.age` method (accessor)
  Can be written alternatively as `$.age`
- `$!age` is a direct call to the variable

8.4.2. Private methods

Normal methods can be called on objects from outside the class.

Private methods are methods that can only be called from within the class.
A possible use case would be a method that calls another one for specific action. The method that interfaces with the outside world is public while the one referenced should stay private. We do not want users to call it directly, so we declare it as private.

The declaration of a private method requires the use of the `!` twigil before its name.
Private methods are called with `!` instead of `.`

```
method !iamprivate {
  #code goes in here
}

method iampublic {
  self!iamprivate;
  #do additional things
}
```

8.5. Class Attributes

Class attributes are attributes that belong to the class itself and not to its objects.
They can be initialized during definition.
Class attributes are declared using `my` instead of `has`.
They are called on the class itself instead of its objects.

```
class Human {
  has $.name;
  my $.counter = 0;
```

```
  method new($name) {
    self.bless(:$name);
    Human.counter++;
  }
}
my $a = Human.new('a');
my $b = Human.new('b');

say Human.counter;
```

8.6. Access Type

Until now all the examples that we've seen, used accessors to get information from the objects' attributes.

What if we need to modify the value of an attribute?
We need to label it as *read/write* using the following keywords `is rw`

```
class Human {
  has $.name;
  has $.age is rw;
}
my $john = Human.new(name => 'John', age => 21);
say $john.age;

$john.age = 23;
say $john.age;
```

By default, all attributes are declared as *read only* but you can explicitly do it using `is readonly`

8.7. Inheritance

8.7.1. Introduction

Inheritance is yet another concept of object oriented programming.

When defining classes, soon enough we will realize that some attributes/methods are common to many classes.
Should we duplicate code?
NO! We should use **inheritance**

Let's consider we want to define two classes a class for Human beings and a class for Employees.
Human beings have 2 attributes: name and age.
Employees have 4 attributes: name, age, company and salary

One would be tempted to define the classes as follow:

```
class Human {
  has $.name;
  has $.age;
}

class Employee {
  has $.name;
  has $.age;
  has $.company;
  has $.salary;
}
```

While technically correct the above piece of code is considered conceptually poor.

A better way to write it would be as follow:

```
class Human {
```

```
  has $.name;
  has $.age;
}

class Employee is Human {
  has $.company;
  has $.salary;
}
```

The `is` keyword defines inheritance.
In object oriented parlance we say Employee is a **child** of Human, and Human is a **parent** of Employee.

All child classes inherit the attributes and methods of the parent class, so there is no need to redefine them.

8.7.2. Overriding

Classes inherit all attributes and methods from their parent classes.
There are cases where we need the method in the child class to behave differently than the one inherited.
To achieve this, we redefine the method in the child class.
This concept is called **overriding**.

In the below example, the method `introduce-yourself` is inherited by the Employee class.

```
class Human {
  has $.name;
  has $.age;
  method introduce-yourself {
    say 'Hi i am a human being, my name is ' ~ self.name;
  }
}
```

```
class Employee is Human {
  has $.company;
  has $.salary;
}

my $john = Human.new(name =>'John', age => 23,);
my $jane = Employee.new(name =>'Jane', age => 25, company => 'Acme', salary => 4000);

$john.introduce-yourself;
$jane.introduce-yourself;
```

Overriding works as follow:

```
class Human {
  has $.name;
  has $.age;
  method introduce-yourself {
    say 'Hi i am a human being, my name is ' ~ self.name;
  }
}

class Employee is Human {
  has $.company;
  has $.salary;
  method introduce-yourself {
    say 'Hi i am a employee, my name is ' ~ self.name ~ ' and I work at: ' ~ self.company;
  }

}

my $john = Human.new(name =>'John',age => 23,);
my $jane = Employee.new(name =>'Jane',age => 25,company => 'Acme',salary => 4000);
```

```
$john.introduce-yourself;
$jane.introduce-yourself;
```

Depending of which class the object is, the right method will be called.

8.7.3. Submethods

Submethods are a type of method that are not inherited by child classes.
They are only accessible from the class they were declared in.
They are defined using the `submethod` keyword.

8.8. Introspection

Introspection is the process of getting information about an object properties like its type, or its attributes or its methods.

```
class Human {
  has Str $.name;
  has Int $.age;
  method introduce-yourself {
    say 'Hi i am a human being, my name is ' ~ self.name;
  }
}

class Employee is Human {
  has Str $.company;
  has Int $.salary;
  method introduce-yourself {
    say 'Hi i am a employee, my name is ' ~ self.name ~ ' and I work at: ' ~ self.company;
  }
}
```

```
my $john = Human.new(name =>'John',age => 23,);
my $jane = Employee.new(name =>'Jane',age => 25,company => 'Acme',salary => 4000);

say $john.WHAT;
say $jane.WHAT;
say $john.^attributes;
say $jane.^attributes;
say $john.^methods;
say $jane.^methods;
say $jane.^parents;
if $jane ~~ Human {say 'Jane is a Human'};
```

Introspection is facilitated by:

- `.WHAT` returns the class from which the object has been created.
- `.^attributes` returns a list containing all attributes of the objects.
- `.^methods` returns all methods that can be called on the object.
- `.^parents` returns all parent classes of the class the object belongs.
- `~~` is called the smart-match operator. It evaluates to *True* if the object is created from the class it is being compared against or any of its inheritances.

9. Exception Handling

9.1. Catching Exceptions

Exceptions are a special behavior that happens at runtime when something goes wrong.
We say that exceptions are *thrown*.

Consider the below script that runs correctly:

```
my Str $name;
$name = "Joanna";
say "Hello " ~ $name;
say "How are you doing today?"
```

Output

```
Hello Joanna
How are you doing today?
```

Now consider this script that throws an exception:

```
my Str $name;
$name = 123;
say "Hello " ~ $name;
say "How are you doing today?"
```

Output

```
Type check failed in assignment to $name; expected Str but got Int
   in block <unit> at exceptions.pl6:2
```

You should have remarked that whenever an error occurs (in this case assigning a number to a string variable) the program will stop and other lines of code will not be evaluated, even if correct.

Exception handling is the process of *catching* an exception that has been *thrown* in order for the script to continue working.

```
my Str $name;
try {
  $name = 123;
  say "Hello " ~ $name;
  CATCH {
    default {
      say "Can you tell us your name again, we couldn't find it in the register.";
    }
  }
}
say "How are you doing today?";
```

Output

```
Can you tell us your name again, we couldn't find it in the register.
How are you doing today?
```

Exception handling is done by using a `try-catch` block.

```
try {
  #code goes in here
  #if anything goes wrong, the script will enter the below CATCH block
  #if nothing goes wrong the CATCH block will be ignored
  CATCH {
    default {
      #the code in here will be evaluated only if an exception has been thrown
    }
  }
}
```

The `CATCH` block can be defined the same way a `given` block is defined. This means we can *catch* and handle

differently many types of exceptions.

```
try {
  #code goes in here
  #if anything goes wrong, the script will enter the below CATCH block
  #if nothing goes wrong the CATCH block will be ignored
  CATCH {
    when X::AdHoc { #do something if an exception of type X::AdHoc is thrown }
    when X::IO { #do something if an exception of type X::IO is thrown }
    when X::OS { #do something if an exception of type X::OS is thrown }
    default { #do something if an exception is thrown and doesn't belong to the above types }
  }
}
```

9.2. Throwing Exceptions

In contrast to catching exceptions, Perl 6 also allows you to explicitly throw exceptions.
Two types of exceptions can be thrown:

- ad-hoc exceptions
- typed exceptions

ad-hoc

```
my Int $age = 21;
die "Error !";
```

typed

```
my Int $age = 21;
```

```
X::AdHoc.new(payload => 'Error !').throw;
```

Ad-hoc exceptions are thrown using the `die` subroutine followed by the exception message.

Typed exceptions are objects, hence the use of the `.new()` constructor in the above example.
All typed exceptions descend from class `X` , below are a few examples:

`X::AdHoc` is the simplest exception type

`X::IO` is related to IO errors

`X::OS` is related to OS errors

`X::Str::Numeric` related to trying to coerce a string to a number

Note	For a complete list of exception types and their associated methods go to http://doc.perl6.org/type.html and navigate to types starting with X.

10. Regular Expressions

A regular expression, or *regex* is a sequence of characters that is used for pattern matching.
The easiest way to understand it is to think of it as a pattern.

```
if 'enlightenment' ~~ m/ light / {
    say "enlightenment contains the word light";
}
```

In this example, the smart match operator `~~` is used to check if a string (enlightenment) contains the word (light).
"Enlightenment" is matched against a regex `m/ light /`

10.1. Regex definition

A regular expression can be defined as follows:

- `/light/`
- `m/light/`
- `rx/light/`

Unless specified explicitly, white space is irrelevant, `m/light/` and `m/ light /` are the same.

10.2. Matching characters

Alphanumeric characters and the underscore `_` are written as is.
All other characters have to be escaped using a backslash or surrounded by quotes.

Backslash

```
if 'Temperature: 13' ~~ m/ \: / {
    say "The string provided contains a colon :";
}
```

Single quotes

```
if 'Age = 13' ~~ m/ '=' / {
    say "The string provided contains an equal character = ";
}
```

Double quotes

```
if 'name@company.com' ~~ m/ "@" / {
    say "This is a valid email address because it contains an @ character";
```

```
}
```

10.3. Matching categories of characters

Characters can be classified into categories and we can match against them.
We can also match against the inverse of that category (everything except it):

Category	Regex	Inverse	Regex
Word character (letter, digit or underscore)	\w	Any character except a word character	\W
Digit	\d	Any character except a digit	\D
Whitespace	\s	Any character except a whitespace	\S
Horizontal whitespace	\h	Any character except a horizontal whitespace	\H
Vertical whitespace	\v	Any character except a vertical whitespace	\V
Tab	\t	Any character except a Tab	\T
New line	\n	Any character except a new line	\N

```
if "John123" ~~ / \d / {
  say "This is not a valid name, numbers are not allowed";
```

```
} else {
  say "This is a valid name"
}
if "John-Doe" ~~ / \s / {
  say "This string contains whitespace";
} else {
  say "This string doesn't contain whitespace"
}
```

10.4. Wildcards

Wildcards can also be used in a regex.

The dot `.` means any single character.

```
if 'abc' ~~ m/ a.c / {
    say "Match";
}
if 'a2c' ~~ m/ a.c / {
    say "Match";
}
if 'ac' ~~ m/ a.c / {
    say "Match";
  } else {
    say "No Match";
}
```

10.5. Quantifiers

Quantifiers come after a character and are used to specify how many times we are expecting it.

The question mark `?` means zero or one time.

```
if 'ac' ~~ m/ a?c / {
    say "Match";
  } else {
    say "No Match";
}
if 'c' ~~ m/ a?c / {
    say "Match";
  } else {
    say "No Match";
}
```

The star `*` means zero or multiple times.

```
if 'az' ~~ m/ a*z / {
    say "Match";
  } else {
    say "No Match";
}
if 'aaz' ~~ m/ a*z / {
    say "Match";
  } else {
    say "No Match";
}
if 'aaaaaaaaaaz' ~~ m/ a*z / {
    say "Match";
  } else {
    say "No Match";
}
if 'z' ~~ m/ a*z / {
    say "Match";
```

```
  } else {
    say "No Match";
}
```

The `+` means at least one time.

```
if 'az' ~~ m/ a+z / {
    say "Match";
  } else {
    say "No Match";
}
if 'aaz' ~~ m/ a+z / {
    say "Match";
  } else {
    say "No Match";
}
if 'aaaaaaaaaaz' ~~ m/ a+z / {
    say "Match";
  } else {
    say "No Match";
}
if 'z' ~~ m/ a+z / {
    say "Match";
  } else {
    say "No Match";
}
```

11. Perl 6 Modules

Perl 6 is a general purpose programming language. It can be used to tackle a multitude of tasks including: text manipulation, graphics, web, databases, network protocols etc.

Reusability is a very important concept whereby programmers don't have to reinvent the wheel each time they want to do a new task.

Perl 6 allows the creation and redistribution of **modules**. Each module is a packaged set of functionality that can be reused once installed.

Panda is a module management tool that comes with Rakudo.

To install a specific module, type the below command in your terminal:

```
panda install "module name"
```

Note	The Perl 6 modules directory can be found on: http://modules.perl6.org/

11.1. Using Modules

MD5 is a cryptographic hash function that produces a 128-bit hash value.
MD5 has a variety of applications of which encryption of passwords stored in a database. When a new user registers, their credentials are not stored as plain text but rather *hashed*. The rationale behind this is that if the DB gets compromised, the attacker will not be able to know what the passwords are.

Lets say you need a script that generates the MD5 hash of a password in preparation for storing it in the DB.

Luckily there's a Perl 6 module that already implemented the MD5 algorithm. Lets install it:

```
panda install Digest::MD5
```

Now run the below script:

```
use Digest::MD5;
```

```
my $password = "password123";
my $hashed-password = Digest::MD5.new.md5_hex($password);

say $hashed-password;
```

In order to run the `md5_hex()` function that creates hashes, we need to load the required module.
The `use` keyword loads the module for use in the script.

12. Unicode

Unicode is a standard for encoding and representing text, that caters for most writing systems in the world.
UTF-8 is a character encoding capable of encoding all possible characters, or code points, in Unicode.

Characters are defined by a:
Grapheme: Visual representation.
Code point: A number assigned to the character.

12.1. Using Unicode

Lets look at how we can output characters using Unicode

```
say "a";
say "\x0061";
say "\c[LATIN SMALL LETTER A]";
```

The above 3 lines showcase different ways of building a character:

1. Writing the character directly (grapheme)

2. Using `\x` and the code point

3. Using `\c` and the code point name

Now lets output a smiley

```
say "☺";
say "\x263a";
say "\c[WHITE SMILING FACE]";
```

Another example combining two code points

```
say "á";
say "\x00e1";
say "\x0061\x0301";
say "\c[LATIN SMALL LETTER A WITH ACUTE]";
```

The letter `á` can be written:

- using its unique code point `\x00e1`
- or as a combination of the code points of `a` and acute `\x0061\x0301`

Some of the methods that can be used:

```
say "á".NFC;
say "á".NFD;
say "á".uniname;
```

```
Output

 NFC:0x<00e1>
```

```
NFD:0x<0061 0301>
LATIN SMALL LETTER A WITH ACUTE
```

`NFC` returns the unique code point.
`NFD` decomposes the character and return the code point of each part.
`uniname` returns the code point name.

Unicode letters can be used as identifiers:

```
my $Δ = 1;
$Δ++;
say $Δ;
```

13. The community

Much discussion happens on the #perl6 IRC channel. This should be your go to place for any enquiry:
http://perl6.org/community/irc

Stay tuned by reading blog posts that focus on Perl 6:
http://pl6anet.org/ is a Perl 6 blog aggregator.