Raku Guide

Naoum Hankache

Table of Contents

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

Some sections of this document refer to other (more complete and accurate) parts of the Raku 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:

- https://github.com/hankache/rakuguide

Feedback

All feedback is welcomed: naoum@hankache.com

If you liked this work, *Star* the repository on Github.

Translations

- Bulgarian: https://raku.guide/bg
- Chinese: https://raku.guide/zh
- Dutch: https://raku.guide/nl
- French: https://raku.guide/fr
- German: https://raku.guide/de
- Indonesian: https://raku.guide/id
- Italian https://raku.guide/it
- Japanese: https://raku.guide/ja
- Portuguese: https://raku.guide/pt
- Spanish: https://raku.guide/es
- Turkish: https://raku.guide/tr
- Russian: https://raku.guide/ru

Chapter 1. Introduction

1.1. What is Raku

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

Raku motto:

- TMTOWTDI (Pronounced Tim Toady): There is more than one way to do it.

1.2. Jargon

- **Raku**: Is a language specification with a test suite. Implementations that pass the specification test suite are considered Raku.
- **Rakudo**: Is a compiler for Raku.
- **Zef**: Is a Raku module installer.
- **Rakudo Star**: Is a bundle that includes Rakudo, Zef, a collection of Raku modules, and documentation.

1.3. Installing Raku

Linux

To install Rakudo Star, run the following commands from your terminal:

```
mkdir ~/rakudo && cd $_
curl -LJO https://rakudo.org/latest/star/src
tar -xzf rakudo-star-*.tar.gz
mv rakudo-star-*/* .
rm -fr rakudo-star-*

./bin/rstar install

echo "export
PATH=$(pwd)/bin/:$(pwd)/share/perl6/site/bin:$(pwd)/share/perl6/vendor/bin:$(pwd)/shar
e/perl6/core/bin:\$PATH" >> ~/.bashrc
source ~/.bashrc
```

For other options, go to https://rakudo.org/star/source

macOS

Four options are available:

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

- Install with MacPorts: `sudo port install rakudo`
- Get the latest installer (file with .dmg extension) from https://rakudo.org/latest/star/macos

Windows

1. For 64-bit architectures: Get the latest installer (file with .msi extension) from https://rakudo.org/latest/star/win
2. After installation, make sure `C:\rakudo\bin` is in the 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 Raku code

Running Raku code can be done using the REPL (Read-Eval-Print Loop). To do this, open a terminal, type `raku` into the terminal window, and hit [Enter]. This will cause a prompt of `>` to appear. Next, type a line of code and hit [Enter]. The REPL will print out the value of the line. You may then type another line, or type `exit` and hit [Enter] to leave the REPL.

Alternatively, write your code in a file, save it and run it. It is recommended that Raku scripts have a `.raku` file name extension. Run the file by typing `raku filename.raku` into the terminal window and hitting [Enter]. Unlike the REPL, this will not automatically print the result of each line: the code must contain a statement like `say` to print output.

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.

Single lines may also be tried non-interactively on the command-line by typing `raku -e 'your code here'` and hitting [Enter].

Rakudo Star bundles a line editor that helps you get the most out of the REPL.

If you installed plain Rakudo instead of Rakudo Star then you probably don't have line editing features enabled (using the up and down arrows for history, left and right to edit input, TAB completion). Consider running the following command and you shall be all set:

- `zef install Linenoise` would work on Windows, Linux and macOS
- `zef install Readline` if you are on Linux and prefer the *Readline* library

1.5. Editors

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

Comma is the Integrated Development Environment designed specifically for Raku.

The community also uses frequently the following editors:

- Atom: better syntax highlighting enabled using Perl 6 FE
- Visual Studio Code: syntax highlighting and error checking enabled using raku-navigator
- Vim: better syntax highlighting enabled using vim-raku
- Emacs: syntax highlighting enabled using raku-mode
- Nano: syntax highlighting enabled using raku.nanorc

1.6. Hello World!

We shall begin with The `hello world` ritual.

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

that can also be written as:

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

1.7. Syntax overview

Raku is **free form**: Most of the time you are free to use any amount of whitespace, although in certain cases whitespace carries meaning.

Statements are typically a single line of code separated with semicolons:

```
say "Hello" if True;
say "World" if False;
```

Semicolons are not required after the last statement in a file or block of code, but it's good practice to include them anyway.

Blocks can contain a collection of statements. Surround statements with curly braces to create a block:

```
{
    say "First statement in the block.";
    say "Second statement in the block.";
}
```

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 names 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 or 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 other's) 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 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
```

Chapter 2. Operators

2.1. Common Operators

The below table lists the most commonly used 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`
`div`	Infix	Integer Division (rounds down)	`3 div 2`	`1`
`%`	Infix	Modulo	`7 % 4`	`3`
`%%`	Infix	Divisibility	`6 %% 4`	`False`
			`6 %% 3`	`True`
`gcd`	Infix	Greatest common divisor	`6 gcd 9`	`3`
`lcm`	Infix	Least common multiple	`6 lcm 9`	`18`
`==`	Infix	Numeric equal	`9 == 7`	`False`
`!=`	Infix	Numeric not equal	`9 != 7`	`True`
`<`	Infix	Numeric less than	`9 < 7`	`False`
`>`	Infix	Numeric greater than	`9 > 7`	`True`
`<=`	Infix	Numeric less than or equal	`7 <= 7`	`True`
`>=`	Infix	Numeric greater than or equal	`9 >= 7`	`True`
`<=>`	Infix	Numeric three-way comparator	`1 <=> 1.0`	`Same`
			`1 <=> 2`	`Less`
			`3 <=> 2`	`More`
`eq`	Infix	String equal	`"John" eq "John"`	`True`
`ne`	Infix	String not equal	`"John" ne "Jane"`	`True`
`lt`	Infix	String less than	`"a" lt "b"`	`True`
`gt`	Infix	String greater than	`"a" gt "b"`	`False`
`le`	Infix	String less than or equal	`"a" le "a"`	`True`
`ge`	Infix	String greater than or equal	`"a" ge "b"`	`False`

Operator	Type	Description	Example	Result
leg	Infix	String three-way comparator	"a" leg "a"	Same
			"a" leg "b"	Less
			"c" leg "b"	More
cmp	Infix	Smart three-way comparator	"a" cmp "b"	Less
			3.5 cmp 2.6	More
=	Infix	Assignment	my $var = 7	Assigns the value of 7 to the variable $var
~	Infix	String concatenation	9 ~ 7	97
			"Hi " ~ "there"	Hi there
x	Infix	String replication	13 x 3	131313
			"Hello " x 3	Hello Hello Hello
~~	Infix	Smart match	2 ~~ 2	True
			2 ~~ Int	True
			"Raku" ~~ "Raku"	True
			"Raku" ~~ Str	True
			"enlightenment" ~~ /light/	light
++	Prefix	Increment	my $var = 2; ++$var;	Increment the variable by 1 and return the result 3
	Postfix	Increment	my $var = 2; $var++;	Return the variable 2 and then increment it
--	Prefix	Decrement	my $var = 2; --$var;	Decrement the variable by 1 and return the result 1
	Postfix	Decrement	my $var = 2; $var--;	Return the variable 2 and then decrement it
+	Prefix	Coerce the operand to a numeric value	+"3"	3
			+True	1
			+False	0
-	Prefix	Coerce the operand to a numeric value and return the negation	-"3"	-3
			-True	-1
			-False	0
?	Prefix	Coerce the operand to a boolean value	?0	False
			?9.8	True
			?"Hello"	True
			?""	False
			my $var; ?$var;	False
			my $var = 7; ?$var;	True

Operator	Type	Description	Example	Result
`!`	`Prefix`	Coerce the operand to a boolean value and return the negation	`!4`	`False`
`..`	`Infix`	Range Constructor	`0..5`	`Creates a range of the interval [0, 5]` [1]
`..^`	`Infix`	Range Constructor	`0..^5`	`Creates a range of the interval [0, 5)` [1]
`^..`	`Infix`	Range Constructor	`0^..5`	`Creates a range of the interval (0, 5]` [1]
`^..^`	`Infix`	Range Constructor	`0^..^5`	`Creates a range of the interval (0, 5)` [1]
`^`	`Prefix`	Range Constructor	`^5`	`Same as 0..^5 Creates a range of the interval [0, 5)` [1]
`…`	`Infix`	Lazy List Constructor	`0…9999`	`return the elements only if requested`
`|`	`Prefix`	Flattening	`|(0..5)`	`(0 1 2 3 4 5)`
			`|(0^..^5)`	`(1 2 3 4)`

2.2. Reversed Operators

Adding `R` before any operator will have the effect of reversing its operands.

Normal Operation	Result	Reversed Operator	Result
`2 / 3`	`0.666667`	`2 R/ 3`	`1.5`
`2 - 1`	`1`	`2 R- 1`	`-1`

2.3. Reduction Operators

Reduction operators work on lists of values. They are formed by surrounding the operator with brackets `[]`

Normal Operation	Result	Reduction Operator	Result
`1 + 2 + 3 + 4 + 5`	`15`	`[+] 1,2,3,4,5`	`15`
`1 * 2 * 3 * 4 * 5`	`120`	`[*] 1,2,3,4,5`	`120`

For the complete list of operators, including their precedence, go to https://docs.raku.org/language/operators

[1] Notations for intervals: https://en.wikipedia.org/wiki/Interval_(mathematics)#Notations_for_intervals

Chapter 3. Variables

Raku 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

This guide offers a simplified model of variables suitable for learning the basics of Raku. For a deeper understanding of variables, see https://docs.raku.org/language/containers

3.1. Scalars

A scalar holds one value or reference.

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

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

A specific set of operations can be performed on a scalar, depending on the value it holds.

String

```
my $name = 'John Doe';
say $name.uc;
say $name.chars;
say $name.flip;
```

```
JOHN DOE
8
eoD nhoJ
```

For the complete list of methods applicable to Strings, see https://docs.raku.org/type/Str

Integer

```
my $age = 17;
say $age.is-prime;
```

```
True
```

For the complete list of methods applicable to Integers, see https://docs.raku.org/type/Int

Rational Number

```
my $age = 2.3;
say $age.numerator;
say $age.denominator;
say $age.nude;
```

```
23
10
(23 10)
```

For the complete list of methods applicable to Rational Numbers, see https://docs.raku.org/type/Rat

3.2. Arrays

Arrays are lists containing multiple values.

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

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

The tilde ~ is used for string 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 one or more elements to the array.
We can access a specific element in the array by specifying its position `@animals[0]`.
`.pop` removes the last element from the array and returns it.
`.splice(a,b)` will remove `b` elements starting at position `a`.

3.2.1. Fixed-size arrays

A basic array is declared as following:

```
my @array;
```

The basic array can have indefinite length and thus is called auto-extending.
The array will accept any number of values with no restriction.

In contrast, we can also create fixed-size arrays.
These arrays cannot be accessed beyond their defined size.

To declare an array of fixed size, specify its maximum number of elements in square brackets immediately after its name:

```
my @array[3];
```

This array will be able to hold a maximum of 3 values, indexed from 0 to 2.

```
my @array[3];
@array[0] = "first value";
@array[1] = "second value";
@array[2] = "third value";
```

You will not be able to add a fourth value to this array:

```
my @array[3];
@array[0] = "first value";
@array[1] = "second value";
@array[2] = "third value";
@array[3] = "fourth value";
```

```
Index 3 for dimension 1 out of range (must be 0..2)
```

3.2.2. Multidimensional arrays

The arrays we saw until now are one-dimensional.
Fortunately, we can define multi-dimensional arrays in Raku.

```
my @tbl[3;2];
```

This array is two-dimensional. The first dimension can have a maximum of 3 values and the second dimension a maximum of 2 values.

Think of it as a 3x2 grid.

```
my @tbl[3;2];
@tbl[0;0] = 1;
@tbl[0;1] = "x";
@tbl[1;0] = 2;
@tbl[1;1] = "y";
@tbl[2;0] = 3;
@tbl[2;1] = "z";
say @tbl
```

```
[[1 x] [2 y] [3 z]]
```

Visual representation of the array:

```
[1 x]
[2 y]
[3 z]
```

For the complete Array reference, see https://docs.raku.org/type/Array

3.3. Hashes

A Hash is a set of Key/Value pairs.

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

Another succinct way of filling the hash:

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

Some of the methods that can be called on hashes are:

Script

```
my %capitals = UK => 'London', Germany => 'Berlin';
%capitals.push: (France => 'Paris');
say %capitals.kv;
say %capitals.keys;
say %capitals.values;
say "The capital of France is: " ~ %capitals<France>;
```

Output

```
(France Paris Germany Berlin UK London)
(France Germany UK)
(Paris Berlin London)
The capital of France is: Paris
```

Explanation

`.push: (key => 'Value')` adds a new key/value pair.
`.kv` returns a list containing all keys and values.
`.keys` returns a list that contains all keys.
`.values` returns a list that contains all values.
We can access a specific value in the hash by specifying its key `%hash<key>`

For the complete Hash reference, see https://docs.raku.org/type/Hash

3.4. Types

In the previous examples, we did not specify what type of values the variables should hold.

`.WHAT` will return the type of value held in a 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 value in `$var` was once (Str) and then (Int).

This style of programming is called dynamic typing. Dynamic in the sense that variables may contain values of Any type.

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

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

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

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

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

Arrays and hashes can also be statically typed:

```
my Int @array = 1,2,3;
say @array;
say @array.WHAT;

my Str @multilingual = "Hello","Salut","Hallo","□□","□□□□□□","□□□□□□";
say @multilingual;
say @multilingual.WHAT;

my Str %capitals = UK => 'London', Germany => 'Berlin';
say %capitals;
say %capitals.WHAT;

my Int %country-codes = UK => 44, Germany => 49;
say %country-codes;
say %country-codes.WHAT;
```

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 Raku 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;    # (Int)
my $var2;
say $var2.WHAT;   # (Any)
$var2 = 1;
say $var2.WHAT;   # (Int)
$var2 = "Hello";
say $var2.WHAT;   # (Str)
$var2 = True;
say $var2.WHAT;   # (Bool)
$var2 = Nil;
say $var2.WHAT;   # (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 Raku. We've been using `my`, so far.

```
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 Raku is delimited by `{ }`. If no block is found, the variable will be available in the whole Raku 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 used in another block.

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

3.7. Assignment vs. Binding

We've seen in the previous examples, how to **assign** values to variables.
Assignment is done using the `=` operator.

```
my Int $var = 123;
say $var;
```

We can change the value assigned to a variable:

Assignment

```
my Int $var = 123;
say $var;
$var = 999;
say $var;
```

Output

```
123
999
```

On the other hand, we cannot change the value **bound** to a variable.
Binding is done using the `:=` operator.

Binding

```
my Int $var := 123;
say $var;
$var = 999;
say $var;
```

Output

```
123
Cannot assign to an immutable value
```

Variables can also be bound to other variables:

```
my $a;
my $b;
$b := $a;
$a = 7;
say $b;
$b = 8;
say $a;
```

Output

```
7
8
```

Binding variables is bi-directional.
`$a := $b` and `$b := $a` have the same effect.

For more info on variables, see https://docs.raku.org/language/variables

Chapter 4. Functions and mutators

It is important to differentiate between functions and mutators.
Functions do not change the state of the object they were called on.
Mutators modify the state of the object.

Script

```
1 my @numbers = [7,2,4,9,11,3];
2
3 @numbers.push(99);
4 say @numbers;      #1
5
6 say @numbers.sort; #2
7 say @numbers;      #3
8
9 @numbers.=sort;
10 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)

Chapter 5. Loops and conditions

Raku has many conditional and looping constructs.

5.1. if

The code runs only if a condition has been met; i.e., an expression evaluates to `True`.

```
my $age = 19;

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

In Raku, 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 specify alternate blocks for execution by 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 Raku 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` behaves 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` and then performed the operation `*100` on each array item.

5.5. given

`given` is the Raku equivalent of the switch statement in other languages, but much more powerful.

```
my $var = 42;

given $var {
    when 0..50 { say 'Less than or equal to 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 Raku to continue matching even after a successful match.

```
my $var = 42;

given $var {
    when 0..50 { say 'Less than or equal to 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.

Raku belongs to the C-family languages.

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

For more info on loops and conditions, see https://docs.raku.org/language/control

Chapter 6. I/O

In Raku, 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 newline 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 behaves like `say` but doesn't add a new line.

Try replacing `say` with `print` and compare the 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 Raku";
```

When the above code runs, the terminal will be waiting for you to input your name. Enter it and then hit [Enter]. Subsequently, it will greet you.

6.1.4. prompt

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

The above example can be written like this:

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

say "Dear $name welcome to Raku";
```

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. It is platform and shell dependent. All shell meta characters are interpreted by the shell, including pipes, redirects, environment variable substitutions and so on.

Run this if you're on Linux/macOS

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

Run this if you're on Windows

```
shell "dir";
```

`echo` and `ls` are common shell keywords on Linux:
`echo` prints text to the terminal (the equivalent of `say` in Raku)
`ls` lists all files and folders in the current directory

`dir` is the equivalent of `ls` on Windows.

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

Raku can list the contents of a directory without resorting to shell commands (by using `ls`, for example).

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

In addition, you can create and delete directories.

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

`mkdir` creates a new directory.
`rmdir` deletes an empty directory and returns an error if not empty.

You can also check if a path exists; 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 raku file `script123.raku`

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

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

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

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

Windows users can use `/` or `\\` to define directories

```
C:\\rakudo\\bin
C:/rakudo/bin
```

For more info on I/O, see https://docs.raku.org/type/IO

Chapter 7. Subroutines

7.1. Definition

Subroutines (also called **subs** or **functions**) are a means of packaging and reusing 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

Subroutines can require input. That input is provided by **arguments**. A subroutine may define zero or more **parameters**. The number and type of parameters that a subroutine defines 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. Multiple dispatch

It is possible to define multiple subroutines that have the same name but different signatures. When the subroutine is called, the runtime environment will decide which version to use based on the number and type of supplied arguments. This type of subroutine is defined the same way as normal subs except that we use the `multi` keyword instead of `sub`.

```
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 Parameters

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

Raku provides us the ability to define subroutines with:

- Optional Parameters
- Default Parameters

Optional parameters are defined by appending `?` to the parameter name.

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

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

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

7.5. Returning values

All the subroutines we've seen so far **do something** — they display some text on the terminal.

Sometimes, though, we execute a subroutine for its **return** value so we can use it later in the flow of our program.

If a function is allowed to run through it's block to the end, the last statement or expression will

determine the return value.

Implicit return

```
sub squared ($x) {
  $x ** 2;
}
say "7 squared is equal to " ~ squared(7);
```

For the sake of clarity, it might be a good idea to *explicitly* specify what we want returned. This can be done using the `return` keyword.

Explicit return

```
sub squared ($x) {
  return $x ** 2;
}
say "7 squared is equal to " ~ squared(7);
```

7.5.1. Restricting return values

In one of the previous examples, we saw how we can restrict the accepted argument to be of a certain type. The same can be done with return values.

To restrict the return value to a certain type, we use the arrow notation `-->` in the signature.

Indicating return type

```
sub squared ($x --> Int) {
  return $x ** 2;
}
say "1.2 squared is equal to " ~ squared(1.2);
```

If we fail to provide a return value that matches the type constraint, an error will be thrown.

```
Type check failed for return value; expected Int but got Rat (1.44)
```

Not only can type constraints control the type of the return value; they can also control its definedness.

In the previous examples, we specified that the return value should be an `Int`.

We could also have specified that the returned `Int` should be strictly defined or undefined using the following signatures:
`-> Int:D` and `-> Int:U`

That being said, it is good practice to use those type constraints.
Below is the modified version of the previous example that uses `:D` to force the returned `Int` to be defined.

```
sub squared ($x --> Int:D) {
  return $x ** 2;
}
say "1.2 squared is equal to " ~ squared(1.2);
```

For more info on subroutines and functions, see https://docs.raku.org/language/functions

Chapter 8. Functional Programming

In this chapter we will take a look at some of the features that facilitate Functional Programming.

8.1. Functions are first-class citizens

Functions/subroutines are first-class citizens:

- They can be passed as arguments
- They can be returned from other functions
- They can be assigned to variables

A great example is the `map` function.
`map` is a *higher order function*, it can accept another function as an argument.

Script

```
my @array = <1 2 3 4 5>;
sub squared($x) {
  $x ** 2
}
say map(&squared,@array);
```

Output

```
(1 4 9 16 25)
```

Explanation

We defined a subroutine called `squared` that takes an argument and multiplies that argument by itself.
Next, we used `map`, a higher order function, and gave it two arguments, the `squared` subroutine and an array.
The result is a list of the squared elements of the array.

Notice that when passing a subroutine as an argument, we need to prepend `&` to its name.

8.2. Anonymous functions

An **anonymous function** is also called a **lambda**.
An anonymous function is not bound to an identifier (it has no name).

Let's rewrite the `map` example and have it use an anonymous function

```
my @array = <1 2 3 4 5>;
say map(-> $x {$x ** 2},@array);
```

Notice that instead of declaring the squared subroutine and passing it as an argument to `map`, we defined it within the anonymous subroutine as `-> $x {$x ** 2}`.

In Raku parlance, we call this notation a **pointy block**

A pointy block may also be used to assign functions to variables:

```
my $squared = -> $x {
  $x ** 2
}
say $squared(9);
```

8.3. Chaining

In Raku, methods can be chained, so you're not required to pass the result of one method to another as an argument.

To illustrate: Given an array, you may need to return the unique values of the array, sorted from biggest to smallest.

Here's a non-chained solution:

```
my @array = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
my @final-array = reverse(sort(unique(@array)));
say @final-array;
```

Here, we call `unique` on `@array`, pass the result as an argument to `sort`, and then pass that result to `reverse`.

In contrast, with chained methods, the above example can be rewritten as:

```
my @array = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
my @final-array = @array.unique.sort.reverse;
say @final-array;
```

You can already see that chaining methods is *easier on the eye*.

8.4. Feed Operator

The **feed operator**, called *pipe* in some functional programming languages, further illustrates method chaining.

Forward Feed

```
my @array = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
@array ==> unique()
       ==> sort()
       ==> reverse()
       ==> my @final-array;
say @final-array;
```

Explanation

```
Start with `@array` then return a list of unique elements
                    then sort it
                    then reverse it
                    then store the result in @final-array
```

Note that the flow of the method calls is top-down — from first to final step.

Backward Feed

```
my @array = <7 8 9 0 1 2 4 3 5 6 7 8 9>;
my @final-array-v2 <== reverse()
                   <== sort()
                   <== unique()
                   <== @array;
say @final-array-v2;
```

Explanation

The backward feed is like the forward feed, but in reverse.
The flow of the method calls is bottom-up — from final to first step.

8.5. Hyper operator

The **hyper operator** `>>.` will call a method on all elements of a list and return a list of the results.

```
my @array = <0 1 2 3 4 5 6 7 8 9 10>;
sub is-even($var) { $var %% 2 };

say @array>>.is-prime;
say @array>>.&is-even;
```

Using the hyper operator we can call methods already defined in Raku, e.g. `is-prime` that tells us if a number is prime or not.
In addition we can define new subroutines and call them using the hyper operator. In this case we have to prepend `&` to the name of the method; e.g., `&is-even`.

This is very practical as it relieves us from writing a `for` loop to iterate over each value.

Raku guarantees that the order of the results is the same as that of the original list. However, there is **no guarantee** that Raku will actually call the methods in list order or in the same thread. So, be careful with methods that have side-effects, such as `say` or `print`.

8.6. Junctions

A **junction** is a logical superposition of values.

In the below example `1|2|3` is a junction.

```
my $var = 2;
if $var == 1|2|3 {
  say "The variable is 1 or 2 or 3"
}
```

The use of junctions usually triggers **autothreading**; the operation is carried out for each junction element, and all the results are combined into a new junction and returned.

8.7. Lazy Lists

A **lazy list** is a list that is lazily evaluated.
Lazy evaluation delays the evaluation of an expression until required, and avoids repeating evaluations by storing results in a lookup table.

The benefits include:

- Performance increase by avoiding needless calculations
- The ability to construct potentially infinite data structures
- The ability to define control flow

To build a lazy list we use the infix operator `...`
A lazy list has **initial element(s)**, a **generator** and an **endpoint**.

Simple lazy list

```
my $lazylist = (1 ... 10);
say $lazylist;
```

The initial element is 1 and the endpoint is 10. No generator was defined so the default generator is the successor (+1)
In other words this lazy list may return (if requested) the following elements (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Infinite lazy list

```
my $lazylist = (1 ... Inf);
say $lazylist;
```

This list may return (if requested) any integer between 1 and infinity, in other words any integer number.

Lazy list built using a deduced generator

```
my $lazylist = (0,2 ... 10);
say $lazylist;
```

The initial elements are 0 and 2 and the endpoint is 10. No generator was defined, but using the initial elements, Raku will deduce that the generator is (+2)
This lazy list may return (if requested) the following elements (0, 2, 4, 6, 8, 10)

Lazy list built using a defined generator

```
my $lazylist = (0, { $_ + 3 } ... 12);
say $lazylist;
```

In this example, we defined explicitly a generator enclosed in `{ }`
This lazy list may return (if requested) the following elements (0, 3, 6, 9, 12)

When using an explicit generator, the endpoint must be one of the values that the generator can return.
If we reproduce the above example with the endpoint being 10 instead of 12, it will not stop. The generator *jumps over* the endpoint.

Alternatively you can replace `0 ⋯ 10` with `0 ⋯^ * > 10`
You can read it as: From 0 until the first value greater than 10 (excluding it)

This will not stop the generator

```
my $lazylist = (0, { $_ + 3 } ... 10);
say $lazylist;
```

This will stop the generator

```
my $lazylist = (0, { $_ + 3 } ...^ * > 10);
say $lazylist;
```

8.8. Closures

All code objects in Raku are closures, which means they can reference lexical variables from an outer scope.

```
sub generate-greeting {
    my $name = "John Doe";
    sub greeting {
      say "Good Morning $name";
    };
    return &greeting;
}
my $generated = generate-greeting;
$generated();
```

If you run the above code, it will display `Good Morning John Doe` on the terminal.
While the result is fairly simple, what is interesting about this example, is that the `greeting` inner subroutine was returned from the outer subroutine before being executed.

`$generated` has become a **closure**.

A **closure** is a special kind of object that combines two things:

- A Subroutine
- The Environment in which that subroutine was created.

The environment consists of any local variable that was in-scope at the time that the closure was created. In this case, `$generated` is a closure that incorporates both the `greeting` subroutine and the `John Doe` string that existed when the closure was created.

Let's take a look at a more interesting example.

```
sub greeting-generator($period) {
  return sub ($name) {
    return "Good $period $name"
  }
}
my $morning = greeting-generator("Morning");
my $evening = greeting-generator("Evening");

say $morning("John");
say $evening("Jane");
```

In this example, we have defined a subroutine `greeting-generator($period)` that accepts a single argument `$period` and returns a new subroutine. The subroutine it returns accepts a single argument `$name` and returns the constructed greeting.

Basically, `greeting-generator` is a subroutine factory. In this example, we used `greeting-generator` to create two new subroutines, one that says `Good Morning` and one that says `Good Evening`.

`$morning` and `$evening` are both closures. They share the same subroutine body definition, but store different environments.
In `$morning` 's environment `$period` is `Morning`. In `$evening` 's environment `$period` is `Evening`.

Chapter 9. Classes & Objects

In the previous chapter, we learned how Raku facilitates Functional Programming.
In this chapter we will take a look at Object Oriented programming in Raku.

9.1. 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** is a template for creating **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 {

}
```

9.2. 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 called **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 accessor. It lets us access the value of the variable without getting direct access to it.

Encapsulation is facilitated in Raku 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.

Therefore, we should rewrite the above class as:

```
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 $john.age;`
It will return this error: `Method 'age' not found for invocant of class 'Human'` because `$!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 observe the result of `say $john.age;`

9.3. Named vs. Positional Parameters

In Raku, 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**.
In our example above, notice that 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 an 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;
```

9.4. Methods

9.4.1. 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 are 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 direct 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

9.4.2. 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
}
```

9.5. 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) {
    Human.counter++;
    self.bless(:$name);
  }
}
my $a = Human.new('a');
my $b = Human.new('b');

say Human.counter;
```

9.6. Access Type

Until now, all the examples that we've seen have 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 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`

9.7. Inheritance

9.7.1. Introduction

Inheritance is 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:

```
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 this would be:

```
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 that 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.

9.7.2. 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 like this:

```
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.

9.7.3. 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.

9.8. Multiple Inheritance

Multiple inheritance is allowed in Raku. A class can inherit from multiple other classes.

```
class bar-chart {
  has Int @.bar-values;
  method plot {
    say @.bar-values;
  }
}

class line-chart {
  has Int @.line-values;
  method plot {
    say @.line-values;
  }
}

class combo-chart is bar-chart is line-chart {
}

my $actual-sales = bar-chart.new(bar-values => [10,9,11,8,7,10]);
my $forecast-sales = line-chart.new(line-values => [9,8,10,7,6,9]);

my $actual-vs-forecast = combo-chart.new(bar-values => [10,9,11,8,7,10],
                                         line-values => [9,8,10,7,6,9]);
say "Actual sales:";
$actual-sales.plot;
say "Forecast sales:";
$forecast-sales.plot;
say "Actual vs Forecast:";
$actual-vs-forecast.plot;
```

Output

```
Actual sales:
[10 9 11 8 7 10]
Forecast sales:
[9 8 10 7 6 9]
Actual vs Forecast:
[10 9 11 8 7 10]
```

Explanation

The `combo-chart` class should be able to hold two series, one for the actual values plotted on bars, and another for forecast values plotted on a line.
This is why we defined it as a child of `line-chart` and `bar-chart`.
You should have noticed that calling the method `plot` on the `combo-chart` didn't yield the required result. Only one series was plotted.
Why did this happen?
`combo-chart` inherits from `line-chart` and `bar-chart`, and both of them have a method called `plot`. When we call that method on `combo-chart` Raku internals will try to resolve the conflict by calling one of the inherited methods.

Correction

In order to behave correctly, we should have overridden the method `plot` in the `combo-chart`.

```
class bar-chart {
  has Int @.bar-values;
  method plot {
    say @.bar-values;
  }
}

class line-chart {
  has Int @.line-values;
  method plot {
    say @.line-values;
  }
}

class combo-chart is bar-chart is line-chart {
  method plot {
    say @.bar-values;
    say @.line-values;
  }
}

my $actual-sales = bar-chart.new(bar-values => [10,9,11,8,7,10]);
my $forecast-sales = line-chart.new(line-values => [9,8,10,7,6,9]);

my $actual-vs-forecast = combo-chart.new(bar-values => [10,9,11,8,7,10],
                                         line-values => [9,8,10,7,6,9]);
say "Actual sales:";
$actual-sales.plot;
say "Forecast sales:";
$forecast-sales.plot;
say "Actual vs Forecast:";
$actual-vs-forecast.plot;
```

Output

```
Actual sales:
[10 9 11 8 7 10]
Forecast sales:
[9 8 10 7 6 9]
Actual vs Forecast:
[10 9 11 8 7 10]
[9 8 10 7 6 9]
```

9.9. Roles

Roles are similar to classes in that they are a collection of attributes and methods.

Roles are declared with the keyword `role`. Classes that wish to implement a role, do so using the `does` keyword.

Let's rewrite the multiple inheritance example using roles:

```
role bar-chart {
  has Int @.bar-values;
  method plot {
    say @.bar-values;
  }
}

role line-chart {
  has Int @.line-values;
  method plot {
    say @.line-values;
  }
}

class combo-chart does bar-chart does line-chart {
  method plot {
    say @.bar-values;
    say @.line-values;
  }
}

my $actual-sales = bar-chart.new(bar-values => [10,9,11,8,7,10]);
my $forecast-sales = line-chart.new(line-values => [9,8,10,7,6,9]);

my $actual-vs-forecast = combo-chart.new(bar-values => [10,9,11,8,7,10],
                                         line-values => [9,8,10,7,6,9]);
say "Actual sales:";
$actual-sales.plot;
say "Forecast sales:";
$forecast-sales.plot;
say "Actual vs Forecast:";
$actual-vs-forecast.plot;
```

Run the above script and you will see that results are the same.

By now you're asking yourself: If roles behave like classes, what's their use?
To answer your question, modify the first script used to showcase multiple inheritance, the one where we *forgot* to override the `plot` method.

```
role bar-chart {
  has Int @.bar-values;
  method plot {
    say @.bar-values;
  }
}

role line-chart {
  has Int @.line-values;
  method plot {
    say @.line-values;
  }
}

class combo-chart does bar-chart does line-chart {
}

my $actual-sales = bar-chart.new(bar-values => [10,9,11,8,7,10]);
my $forecast-sales = line-chart.new(line-values => [9,8,10,7,6,9]);

my $actual-vs-forecast = combo-chart.new(bar-values => [10,9,11,8,7,10],
                                         line-values => [9,8,10,7,6,9]);
say "Actual sales:";
$actual-sales.plot;
say "Forecast sales:";
$forecast-sales.plot;
say "Actual vs Forecast:";
$actual-vs-forecast.plot;
```

Output

```
===SORRY!===
Method 'plot' must be resolved by class combo-chart because it exists in multiple
roles (line-chart, bar-chart)
```

Explanation

If multiple roles are applied to the same class and a conflict exists, a compile-time error will be thrown.
This is a much safer approach than multiple inheritance, where conflicts are not considered errors and are simply resolved at runtime.

Roles will warn you that there's a conflict.

9.10. Introspection

Introspection is the process of getting information about an object, like its type, attributes or 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 was created
- `.^attributes` — returns all the attributes of the object
- `.^methods` — returns all the methods that can be called on the object
- `.^parents` — returns the parent classes of the object
- `~~` 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.

For more info on Object Oriented Programming in Raku, see:

- https://docs.raku.org/language/classtut
- https://docs.raku.org/language/objects

Chapter 10. Exception Handling

10.1. 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.raku:2
```

Notice 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.

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 exception of type X::AdHoc is thrown }
    when X::IO      { # do something if exception of type X::IO is thrown }
    when X::OS      { # do something if exception of type X::OS is thrown }
    default         { # do something if exception is thrown and doesn't belong to the
above types }
  }
}
```

10.2. Throwing Exceptions

Raku 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

For a complete list of exception types and their associated methods, go to https://docs.raku.org/type-exceptions.html

Chapter 11. Regular Expressions

A regular expression, or *regex*, is a sequence of characters that is used for pattern matching.
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 the regex `m/ light /`

11.1. Regex definition

A regular expression can be defined like this:

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

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

11.2. 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";
}
```

11.3. 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"
}
```

11.4. Unicode properties

Matching against categories of characters, as seen in the preceding section, is convenient.
That being said, a more systematic approach would be to use Unicode properties.
This allows you to match against categories of characters inside and outside of the ASCII standard.
Unicode properties are enclosed in `<: >`

```
if "Devanagari Numbers ०१२" ~~ / <:N> / {
  say "Contains a number";
} else {
  say "Doesn't contain a number"
}
if "Привет, Иван." ~~ / <:Lu> / {
  say "Contains an uppercase letter";
} else {
  say "Doesn't contain an upper case letter"
}
if "John-Doe" ~~ / <:Pd> / {
  say "Contains a dash";
} else {
  say "Doesn't contain a dash"
}
```

11.5. 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";
}
```

11.6. 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";
}
```

11.7. Match Results

Whenever the process of matching a string against a regex is successful, the match result is stored in a special variable $/

Script

```
if 'Rakudo is a Perl 6 compiler' ~~ m/:s Perl 6/ {
    say "The match is: " ~ $/;
    say "The string before the match is: " ~ $/.prematch;
    say "The string after the match is: " ~ $/.postmatch;
    say "The matching string starts at position: " ~ $/.from;
    say "The matching string ends at position: " ~ $/.to;
}
```

Output

```
The match is: Perl 6
The string before the match is: Rakudo is a
The string after the match is: compiler
The matching string starts at position: 12
The matching string ends at position: 18
```

Explanation

$/ returns a *Match Object* (the string that matches the regex)
The following methods can be called on the *Match Object*:

`.prematch` returns the string preceding the match.
`.postmatch` returns the string following the match.
`.from` returns the starting position of the match.
`.to` returns the ending position of the match.

By default, whitespace in a regex definition is ignored.
If we want to match against a regex containing whitespace, we have to do so explicitly.
The `:s` in the regex `m/:s Perl 6/` forces whitespace to be considered.
Alternatively, we could have written the regex as `m/ Perl\s6 /` and used `\s` which represents a whitespace.
If a regex contains more than a single whitespace, using `:s` is a better option than using `\s` for each and every whitespace.

11.8. Example

Let's check if an email is valid or not.
For the sake of this example we will assume that a valid email address has this format:
first name [dot] last name [at] company [dot] (com/org/net)

The regex used in this example for email validation is not very accurate.
Its sole purpose is to demonstrate regex functionality in Raku.
Do not use it as-is in production.

Script

```
my $email = 'john.doe@perl6.org';
my $regex = / <:L>+\.<:L>+\@<:L+:N>+\.<:L>+ /;

if $email ~~ $regex {
  say $/ ~ " is a valid email";
} else {
  say "This is not a valid email";
}
```

Output

```
john.doe@perl6.org is a valid email
```

Explanation

`<:L>` matches a single letter
`<:L>+` matches one or more letters
`\.` matches a single [dot] character
`\@` matches a single [at] character
`<:L+:N>` matches a letter or a single number
`<:L+:N>+` matches one or more letters or numbers

The regex can be decomposed as following:

- **first name** `<:L>+`
- **[dot]** `\.`
- **last name** `<:L>+`
- **[at]** `\@`
- **company name** `<:L+:N>+`
- **[dot]** `\.`
- **com/org/net** `<:L>+`

Alternatively, a regex can be broken down into multiple named regexes

```
my $email = 'john.doe@perl6.org';
my regex many-letters { <:L>+ };
my regex dot { \. };
my regex at { \@ };
my regex many-letters-numbers { <:L+:N>+ };

if $email ~~ / <many-letters> <dot> <many-letters> <at> <many-letters-numbers> <dot>
<many-letters> / {
  say $/ ~ " is a valid email";
} else {
  say "This is not a valid email";
}
```

A named regex is defined using the following syntax: `my regex regex-name { regex definition }`
A named regex can be called using the following syntax: `<regex-name>`

For more info on regexes, see https://docs.raku.org/language/regexes

Chapter 12. Raku Modules

Raku 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.

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

Zef is a module management tool that comes with Rakudo Star.

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

```
zef install "module name"
```

The Raku modules directory can be found on: https://raku.land/

12.1. Using Modules

MD5 is a cryptographic hash function that produces a 128-bit hash value.
MD5 has a variety of applications, including the encryption of stored passwords 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.

Luckily, you don't need to implement the MD5 algorithm yourself; there's a Raku module already implemented.

Let's install it:
`zef 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.

In practice MD5 hashing alone is not sufficient, because it is prone to dictionary attacks.
It should be combined with a salt https://en.wikipedia.org/wiki/Salt_(cryptography).

Chapter 13. Unicode

Unicode is a standard for encoding and representing text 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.
Code point name: A name assigned to the character.

13.1. Using Unicode

Let's 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;
```

Unicode can be used to do math:

```
my $var = 2 + ;
say $var;
```

13.2. Unicode-aware Operations

13.2.1. Numbers

Arabic numerals are the ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. This numeral set is the most used worldwide.

Nonetheless different sets of numerals are used to a lesser extent in different parts of the world.

No special care needs to be taken when using a numeral set different than the Arabic numerals. All methods/operators work as expected.

```
say (,,,1,2,3).sort; # (1 2 3 4 5 6)
say 1 + ;             # 10
```

13.2.2. Strings

If we were to use generic string operations, we might not always get the result that we were looking for, especially when comparing or sorting.

Comparison

```
say 'a' cmp 'B'; # More
```

The above example shows that `a` is bigger than `B`. The reason being that the code point of lowercase `a` is bigger than the code point of capital `B`.

While technically correct, this is probably not what we were looking for.

Luckily Raku has methods/operators that implement the Unicode Collation Algorithm.
One of them is `unicmp` that behaves like the above showcased `cmp` but is unicode-aware.

```
say 'a' unicmp 'B'; # Less
```

As you can see, using the `unicmp` operator now yields the expected result that `a` is smaller than `B`.

Sorting

As an alternative to the `sort` method that sorts using code points, Raku provides a `collate` method that implements the Unicode Collation Algorithm.

```
say ('a','b','c','D','E','F').sort;    # (D E F a b c)
say ('a','b','c','D','E','F').collate; # (a b c D E F)
```

Chapter 14. Parallelism, Concurrency and Asynchrony

14.1. Parallelism

Under normal circumstances, all tasks in a program run sequentially.
This might not be a problem, unless what you're trying to do takes a lot of time.

Thankfully, Raku has features that will enable you to run things in parallel.
At this stage, it is important to note that parallelism can mean one of two things:

- **Task Parallelism**: Two (or more) independent expressions running in parallel.
- **Data Parallelism**: A single expression iterating over a list of elements in parallel.

Let's begin with the latter.

14.1.1. Data Parallelism

```
my @array = 0..50000;                       # Array population
my @result = @array.map({ is-prime $_ });   # call is-prime for each array element
say now - INIT now;                         # Output the time it took for the script
to complete
```

Considering the above example:

We are only doing one operation `@array.map({ is-prime $_ })`
The `is-prime` subroutine is being called for each array element sequentially:
`is-prime @array[0]` then `is-prime @array[1]` then `is-prime @array[2]` etc.

Fortunately we can call `is-prime` on multiple array elements at the same time:

```
my @array = 0..50000;                         # Array population
my @result = @array.race.map({ is-prime $_ }); # call is-prime for each array element
say now - INIT now;                           # Output the time it took to complete
```

Notice the use of `race` in the expression. This method will enable parallel iteration of the array elements.

After running both examples (with and without `race`), compare the time it took for both scripts to complete.

`race` will not preserve the order of elements. If you wish to do, so use `hyper` instead.

race

```
my @array = 1..1000;
my @result = @array.race.map( {$_ + 1} );
.say for @result;
```

hyper

```
my @array = 1..1000;
my @result = @array.hyper.map( {$_ + 1} );
.say for @result;
```

If you run both examples, you should notice that one is sorted and the other is not.

14.1.2. Task Parallelism

```
my @array1 = 0..49999;
my @array2 = 2..50001;

my @result1 = @array1.map( {is-prime($_ + 1)} );
my @result2 = @array2.map( {is-prime($_ - 1)} );

say @result1 eqv @result2;

say now - INIT now;
```

Considering the above example:

1. We defined 2 arrays
2. applied a different operation for each array and stored the results
3. and checked if both results are the same

The script waits for `@array1.map( {is-prime($_ + 1)} )` to finish
and then evaluates `@array2.map( {is-prime($_ - 1)} )`

Both operations applied to each array do not depend on each other.

Why not do both in parallel?

```
my @array1 = 0..49999;
my @array2 = 2..50001;

my $promise1 = start @array1.map( {is-prime($_ + 1)} ).eager;
my $promise2 = start @array2.map( {is-prime($_ - 1)} ).eager;

my @result1 = await $promise1;
my @result2 = await $promise2;

say @result1 eqv @result2;

say now - INIT now;
```

Explanation

The `start` subroutine evaluates the code and returns **an object of type promise** or shortly **a promise**.
If the code is evaluated correctly, the *promise* will be **kept**.
If the code throws an exception, the *promise* will be **broken**.

The `await` subroutine waits for a **promise**.
If it's **kept** it will get the returned values.
If it's **broken** it will get the exception thrown.

Check the time it took each script to complete.

Parallelism always adds a threading overhead. If that overhead is not offset by gains in computational speed, the script will seem slower.
This is why, using `race`, `hyper`, `start` and `await` for fairly simple scripts can actually slow them down.

14.2. Concurrency and Asynchrony

For more info on Concurrency and Asynchronous Programming, see https://docs.raku.org/language/concurrency

Chapter 15. Native Calling Interface

Raku gives us the ability to use C libraries, using the Native Calling Interface.

`NativeCall` is a standard module that ships with Raku and offers a set of functionality to ease the job of interfacing Raku and C.

15.1. Calling a function

Consider the below C code that defines a function called `hellofromc`. This function prints on the terminal `Hello from C`. It doesn't accept any argument nor return any value.

ncitest.c

```
#include <stdio.h>

void hellofromc () {
  printf("Hello from C\n");
}
```

Depending on your OS run the following commands to compile the above C code into a library.

On Linux:

```
gcc -c -fpic ncitest.c
gcc -shared -o libncitest.so ncitest.o
```

On Windows:

```
gcc -c ncitest.c
gcc -shared -o ncitest.dll ncitest.o
```

On macOS:

```
gcc -dynamiclib -o libncitest.dylib ncitest.c
```

Within the same directory where you compiled your C library, create a new Raku file that contains the below code and run it.

ncitest.raku

```
use NativeCall;

constant LIBPATH = "$*CWD/ncitest";
sub hellofromc() is native(LIBPATH) { * }

hellofromc();
```

Explanation:

First of all we declared that we will be using the `NativeCall` module.
Then we created a constant `LIBPATH` that holds the path to the C library.
Notice that `$*CWD` returns the current working directory.
Then we created a new Raku subroutine called `hellofromc()` that should act as a wrapper to its counterpart C function having the same name and residing in the C library found in `LIBPATH`.
All of this was done by using the `is native` trait.
Finally we called our Raku subroutine.

In essence, it all boils down to declaring a subroutine with the trait `is native` and the name of the C library.

15.2. Renaming a function

In the above part, we saw how we can call a very simple C function by wrapping it with a Raku subroutine having the same name, using the `is native` trait.

In some cases, we would want to change the name of the Raku subroutine.
To do so, we use the `is symbol` trait.

Lets modify the above Raku script and rename the Raku subroutine `hello` instead of `hellofromc`

ncitest.raku

```
use NativeCall;

constant LIBPATH = "$*CWD/ncitest";
sub hello() is native(LIBPATH) is symbol('hellofromc') { * }

hello();
```

Explanation:

In case the Raku subroutine has a different name than its C counterpart, we should use the `is symbol` trait with the name of the original C function.

15.3. Passing Arguments

Compile the following modified C library and run the Raku script found below again.
Notice how we modified both C and Raku code to accept a string (`char*` in C and `Str` in Raku)

ncitest.c

```
#include <stdio.h>

void hellofromc (char* name) {
  printf("Hello, %s! This is C!\n", name);
}
```

ncitest.raku

```
use NativeCall;

constant LIBPATH = "$*CWD/ncitest";
sub hello(Str) is native(LIBPATH) is symbol('hellofromc') { * }

hello('Jane');
```

15.4. Returning values

Lets repeat the process one more time and create a simple calculator that takes 2 integers and add them.
Compile the C library and run the Raku script.

ncitest.c

```
int add (int a, int b) {
  return (a + b);
}
```

ncitest.raku

```
use NativeCall;

constant LIBPATH = "$*CWD/ncitest";
sub add(int32,int32 --> int32) is native(LIBPATH) { * }

say add(2,3);
```

Notice how both C and Raku functions accept two integers and return one (`int` in C and `int32` in Raku)

15.5. Types

You might have asked yourself why did we use `int32` instead of `Int` in the latest Raku script.
Some Raku types like `Int`, `Rat` etc. can't be used as is to pass and receive values from a C function.
One must use in Raku the same types as the ones in C.

Luckily Raku provides many types that map to their respective C counterpart.

C Type	Raku Type
`char`	`int8`
`int8_t`	
`short`	`int16`
`int16_t`	

C Type	Raku Type
`int`	`int32`
`int32_t`	
`int64_t`	`int64`
`unsigned char`	`uint8`
`uint8_t`	
`unsigned short`	`uint16`
`uint16_t`	
`unsigned int`	`uint32`
`uint32_t`	
`uint64_t`	`uint64`
`long`	`long`
`long long`	`longlong`
`float`	`num32`
`double`	`num64`
`size_t`	`size_t`
`bool`	`bool`
`char*` (String)	`Str`
Arrays: For example `int*` (Array of int) and `double*` (Array of double)	`CArray`: For example `CArray[int32]` and `CArray[num64]`

For more info on the Native Calling Interface, see https://docs.raku.org/language/nativecall

Chapter 16. The Community

- #raku IRC channel. Much discussion happens on IRC.
  This should be your go to place for any enquiry for which you want an immediate answer: https://raku.org/community/irc
- StackOverflow Raku questions is a place where questions about Raku can be answered more in-depth.
- Rakudo Weekly a weekly overview of changes in and around Rakudo.
- Planet Raku blog aggregator. Stay tuned by reading blog posts that focus on Raku.
- /r/rakulang subscribe to the Raku subreddit.
- @raku_news follow the community on twitter.