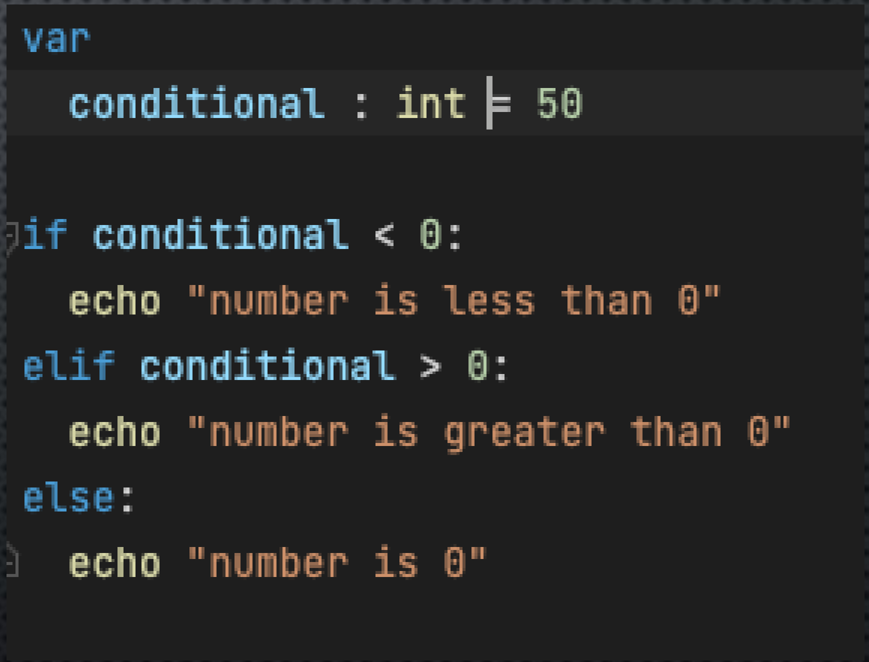
NIM PROGRAMMING LANGUAGE

PRESENTED BY AYMAN ERRARHICHE

What is Nim?

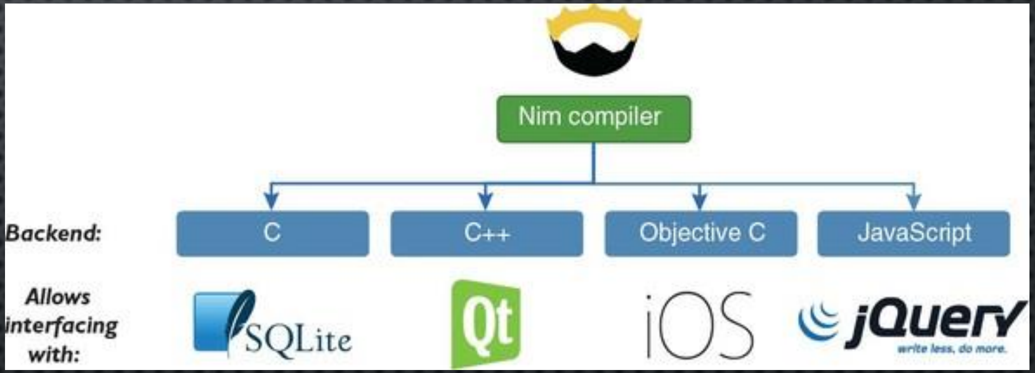
- Nim is a general-purpose programming language designed to be efficient, expressive, and elegant.
- It supports metaprogramming, functional, message passing, procedural, and Object-oriented Programming
- It compiles to C/C++/Objective C and JavaScript.
- Nim was created to be a language as fast as C, as expressive as Python, and as extensible as Lisp.
- Nim can be used for web development, video games, scripting, command line applications, UI applications and a lot more!

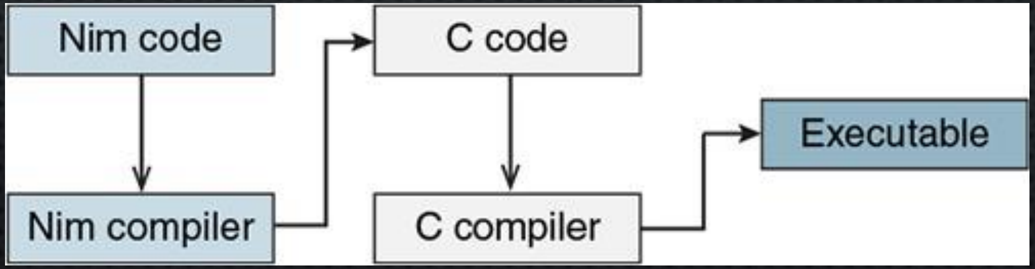


```
var
  conditional : int = 50

if conditional < 0:
  echo "number is less than 0"
elif conditional > 0:
  echo "number is greater than 0"
else:
  echo "number is 0"
```

COMPILATION

- NIM TAKES ADVANTAGE OF THE ASPECTS OF C, INCLUDING ITS PORTABILITY, WIDESPREAD USE, AND EFFICIENCY.
- COMPILING TO C ALSO MAKES IT EASY TO USE EXISTING C AND C++ LIBRARIES. ALL YOU NEED TO DO IS WRITE SOME SIMPLE WRAPPER CODE. YOU CAN WRITE THIS CODE MUCH FASTER BY USING A TOOL CALLED C2NIM. THIS TOOL CONVERTS C AND C++ HEADER FILES TO NIM CODE, WHICH WRAPS THOSE FILES.



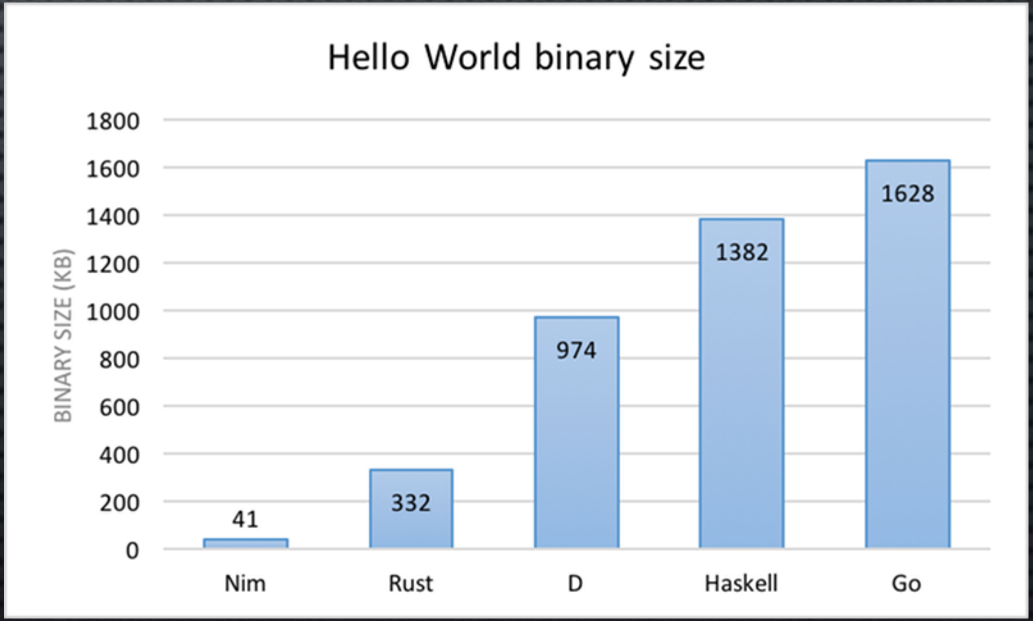


A LITTLE BIT ABOUT NIM'S HISTORY

- Andreas Rumpf started developing Nim in 2005. It was originally named Nimrod when the project was made public in 2008. The project soon gained support and many contributions from the open source community, with many volunteers around the world contributing code via pull requests on GitHub.
- The first version of the Nim compiler was written in Pascal using the Free Pascal compiler. In 2008, a version of the compiler written in Nim was released.

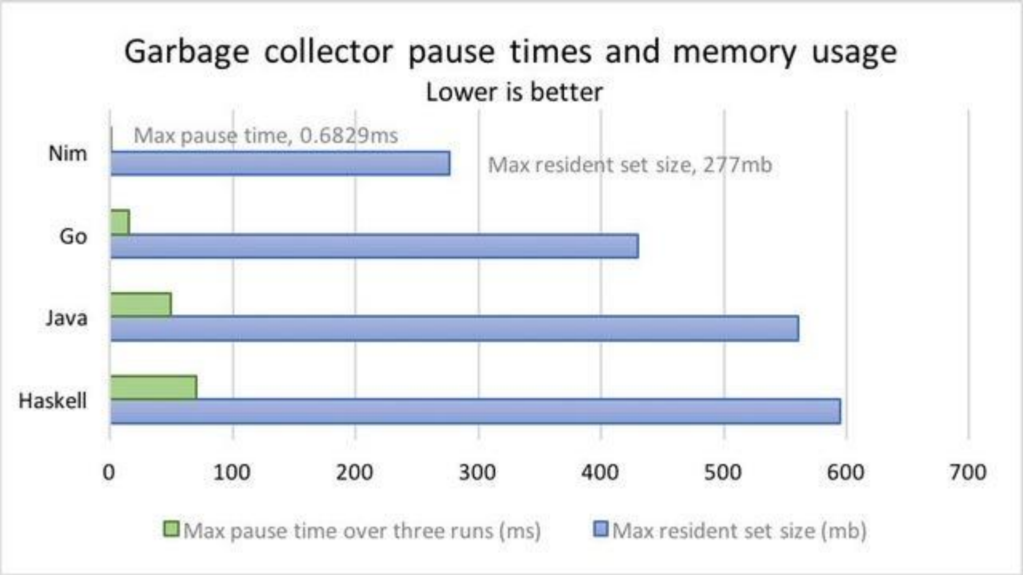
NATIVE PERFORMANCE WITH STATE-OF-THE-ART OPTIMIZATIONS

- By compiling to C, NIM is able to take advantage of many features offered by modern C compilers. The primary benefits gained by this compilation model include incredible portability and optimizations, which have been implemented over more than 40 years by various individuals and influential companies.
- The binaries produced by Nim have zero dependencies and are typically very small. This makes their distribution easy and keeps the users happy.



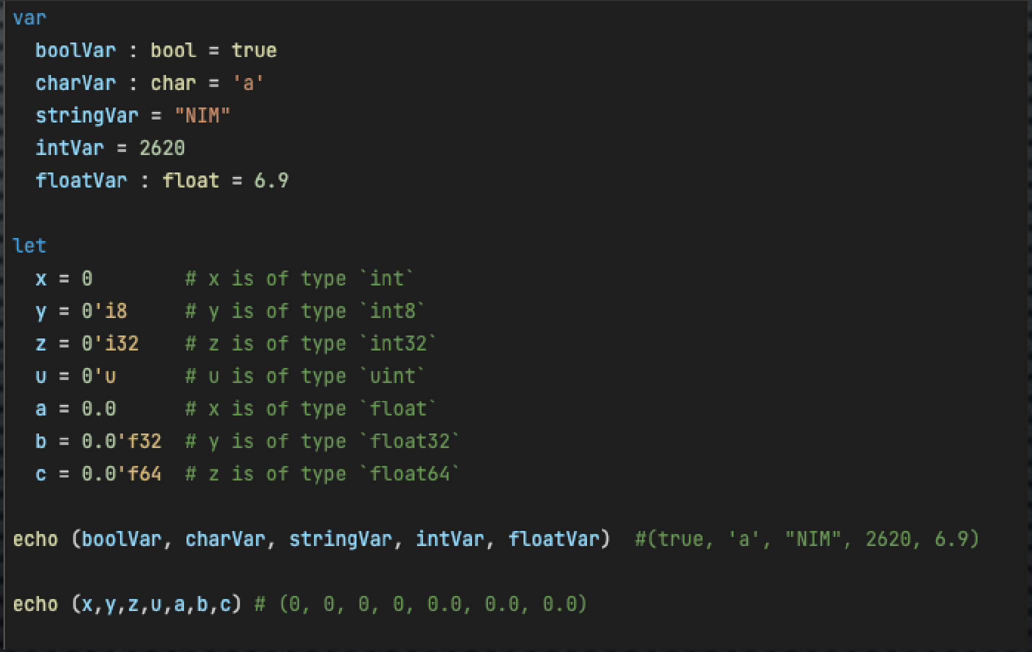
GARBAGE COLLECTOR OPTIONS?

- In NIM you can choose from a deferred reference counting with cycle detection garbage collector that is fast; incremental and causeless; or a soft real-time garbage collector that is deterministic allowing you to specify its max pause time.
- It's optional too!



BASIC DATA TYPES IN NIM

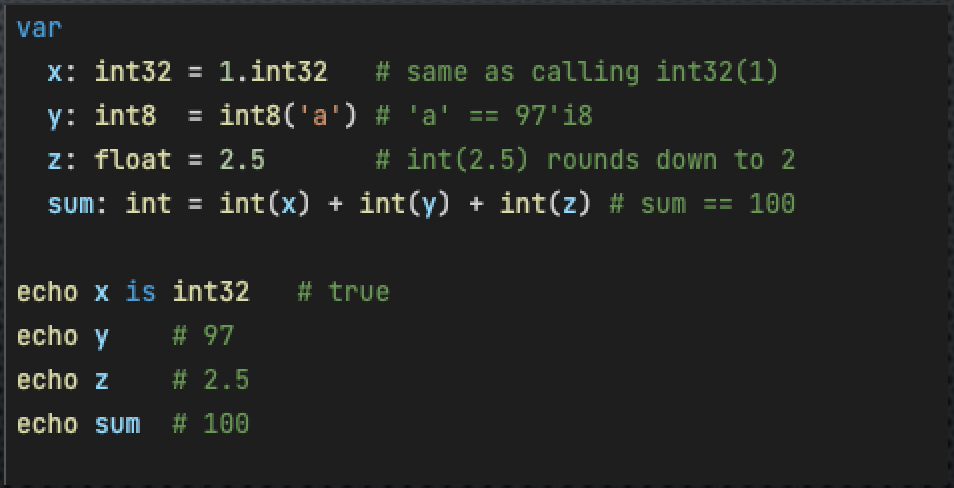
- Bool : true/false (it has logical operators like Python's (e.g and, or, not, xor...)
- Char: enclosed in single quotes can be compared with the ==, <, <=, >, >= operators. The $ operator converts a char to a string.
- String: String variables are mutable, so appending to a string is possible. They are both zero-terminated and have a length field. You can use the & operator to concatenate strings and add to append to a string.
- Int: Nim has these integer types built- in: int, int8, int16, int32, int64, uint, uint8, uint16, uint32 and uint64.
- Floats: Nim has these floating-point types built-in: float float32 float64.



```
var
  boolVar : bool = true
  charVar : char = 'a'
  stringVar = "NIM"
  intVar = 2620
  floatVar : float = 6.9

let
  x = 0       # x is of type `int`
  y = 0'i8    # y is of type `int8`
  z = 0'i32   # z is of type `int32`
  u = 0'u     # u is of type `uint`
  a = 0.0     # x is of type `float`
  b = 0.0'f32 # y is of type `float32`
  c = 0.0'f64 # z is of type `float64`

echo (boolVar, charVar, stringVar, intVar, floatVar)  #(true, 'a', "NIM", 2620, 6.9)

echo (x,y,z,u,a,b,c) # (0, 0, 0, 0, 0.0, 0.0, 0.0)
```

TYPE CONVERSION

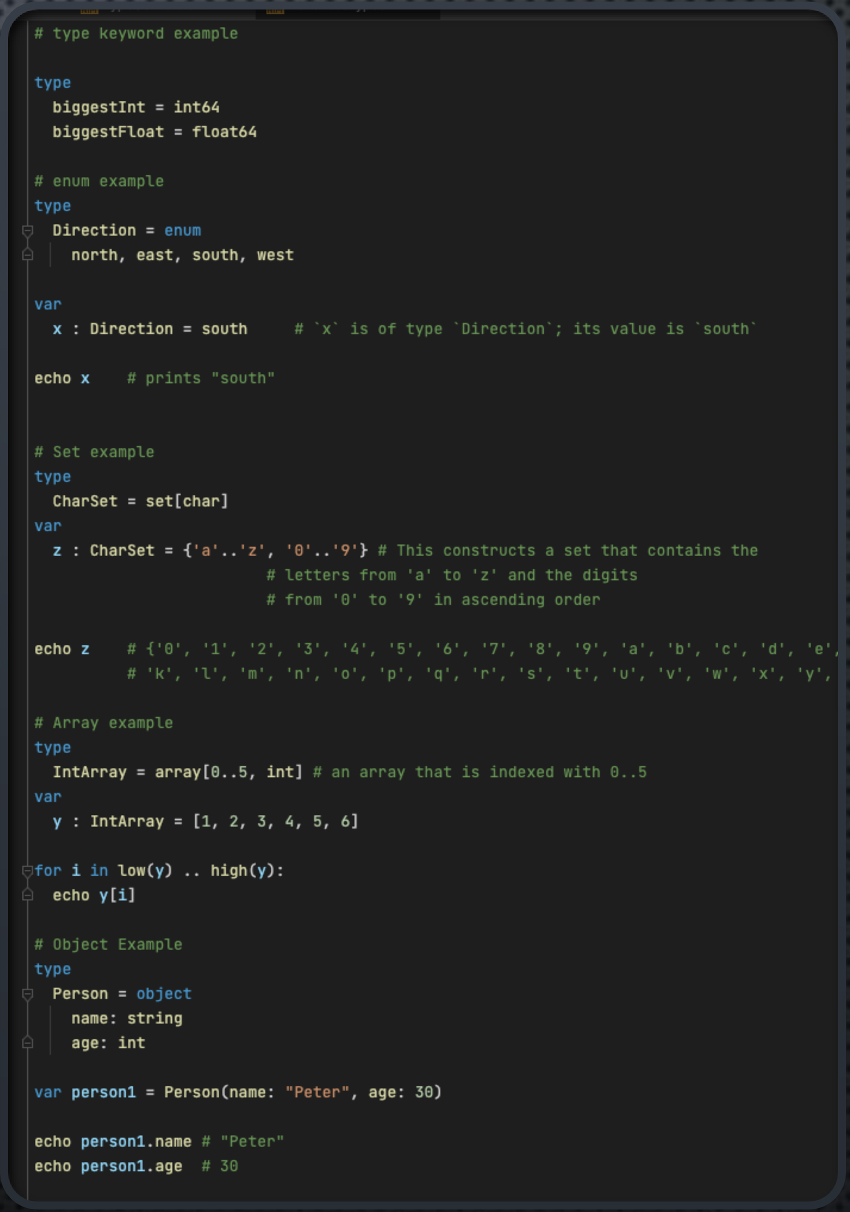
- Conversion between numerical types is performed by using the type as a function:



```
var
  x: int32 = 1.int32   # same as calling int32(1)
  y: int8  = int8('a') # 'a' == 97'i8
  z: float = 2.5       # int(2.5) rounds down to 2
  sum: int = int(x) + int(y) + int(z) # sum == 100

echo x is int32   # true
echo y    # 97
echo z    # 2.5
echo sum  # 100
```

ADVANCED TYPES

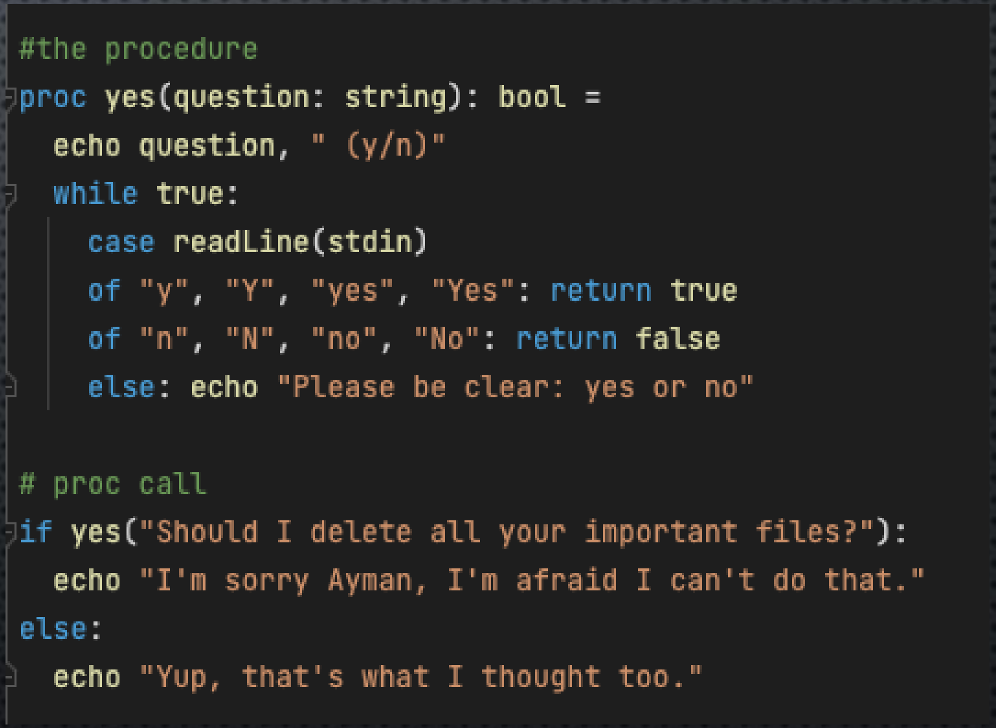
- In Nim new types can be defined within a type statement
- Nim has enums and can assign an enum value to a variable. The $ operator can convert any enumeration value to its name, and the ord proc can convert it to its underlying integer value.
- The set type models the mathematical notion of a set. The set's basetype can only be an ordinal type of a certain size, namely: int8/int16/uint8/uint16 or equivalent.
- An array is a simple fixed-length container. Each element in an array has the same type. The array's index type can be any ordinal type and it can be constructed using [].
- An object is a value type, which means that when an object is assigned to a new variable all its components are copied as well. Each object type Foo has a constructor Foo(field: value, ...) where all of its fields can be initialized.
- It has plenty of other types like tuples, slices, open-array, sequences...



```
# type keyword example

type
  biggestInt = int64
  biggestFloat = float64

# enum example
type
  Direction = enum
    north, east, south, west

var
  x : Direction = south     # `x` is of type `Direction`; its value is `south`

echo x    # prints "south"


# Set example
type
  CharSet = set[char]
var
  z : CharSet = {'a'..'z', '0'..'9'} # This constructs a set that contains the
                         # letters from 'a' to 'z' and the digits
                         # from '0' to '9' in ascending order

echo z    # {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e',
          # 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',

# Array example
type
  IntArray = array[0..5, int] # an array that is indexed with 0..5
var
  y : IntArray = [1, 2, 3, 4, 5, 6]

for i in low(y) .. high(y):
  echo y[i]

# Object Example
type
  Person = object
    name: string
    age: int

var person1 = Person(name: "Peter", age: 30)

echo person1.name # "Peter"
echo person1.age  # 30
```

PROCEDURES / FUNCTIONS

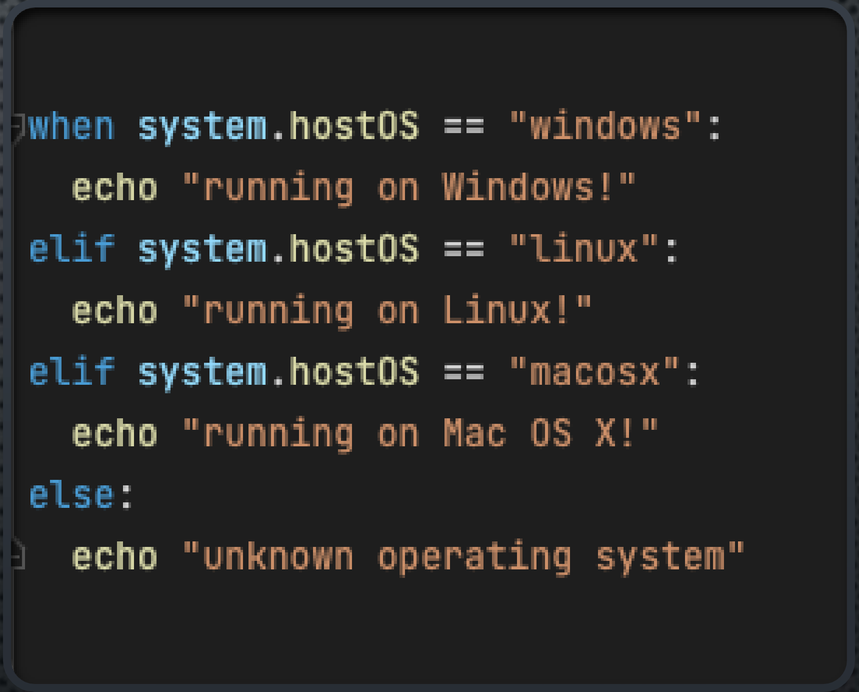
- To define new commands like echo seen in the previous examples, the concept of a ***procedure*** is needed. (Some languages call them *methods* or *functions*.) In Nim new procedures are defined with the **proc** keyword:



```nim
#the procedure
proc yes(question: string): bool =
  echo question, " (y/n)"
  while true:
    case readLine(stdin)
    of "y", "Y", "yes", "Yes": return true
    of "n", "N", "no", "No": return false
    else: echo "Please be clear: yes or no"

# proc call
if yes("Should I delete all your important files?"):
  echo "I'm sorry Ayman, I'm afraid I can't do that."
else:
  echo "Yup, that's what I thought too."
```

Control Flow Statements

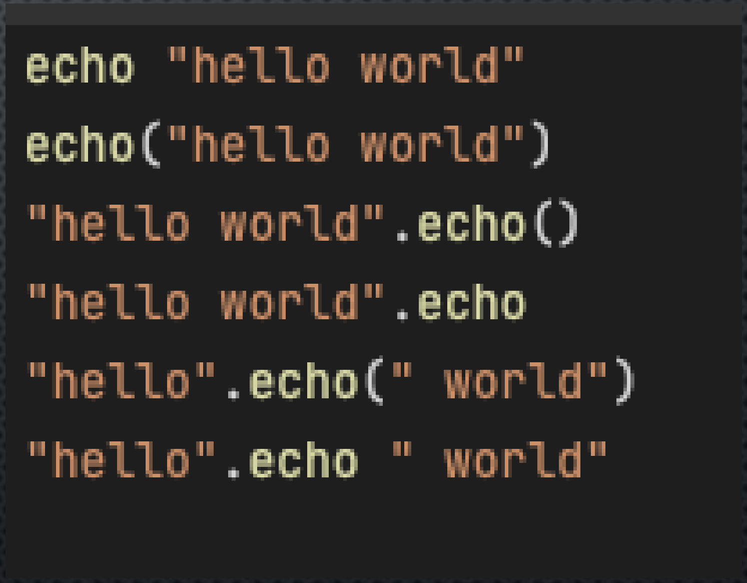
- If,elif,else
- Switch case
- Break and continue
- For loop
- While loop
- The when statement is almost identical to the if statement, but with these differences:
    - Each condition must be a constant expression since it is evaluated by the compiler.
    - The statements within a branch do not open a new scope.
    - The compiler checks the semantics and produces code *only* for the statements that belong to the first condition that evaluates to true.
    - The when statement is useful for writing platform-specific code, similar to the #ifdef construct in the C programming language.



```
when system.hostOS == "windows":
  echo "running on Windows!"
elif system.hostOS == "linux":
  echo "running on Linux!"
elif system.hostOS == "macosx":
  echo "running on Mac OS X!"
else:
  echo "unknown operating system"
```

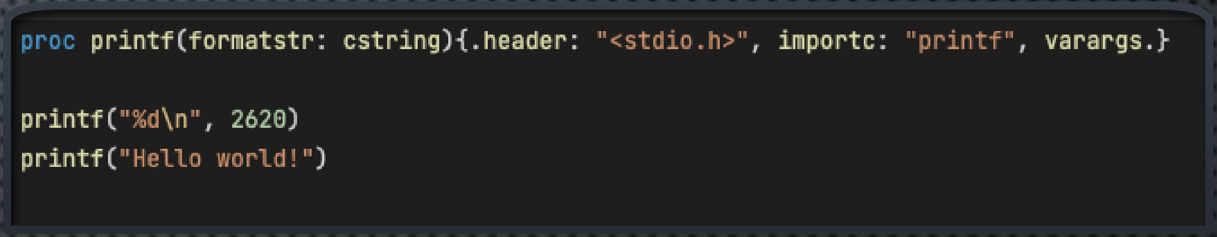
UNIFORM FUNCTION CALL SYNTAX

- Nim supports Uniform function call syntax (UFCS) which provides a large degree of flexibility in use.
- For example, each of these lines does the same call, just with different syntax:

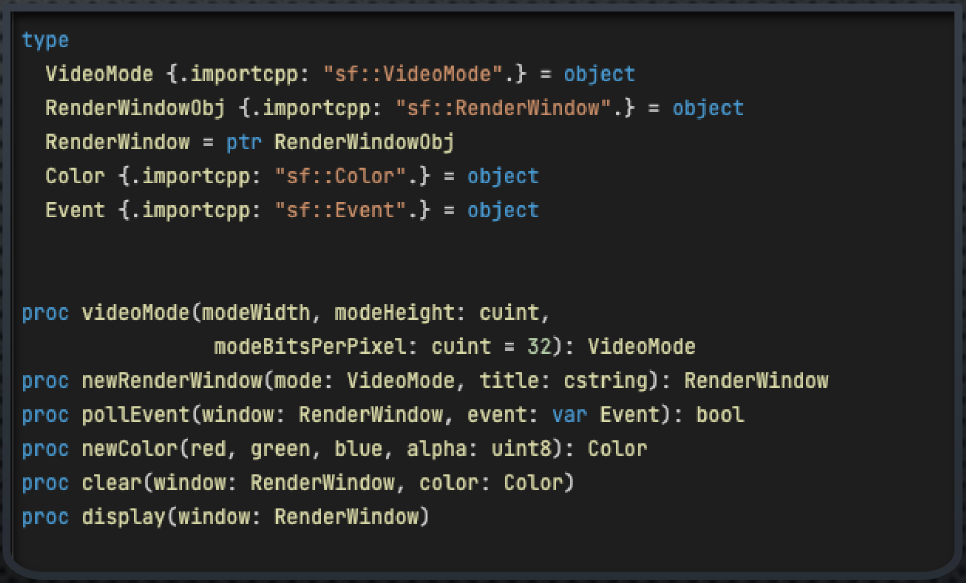


```
echo "hello world"
echo("hello world")
"hello world".echo()
"hello world".echo
"hello".echo(" world")
"hello".echo " world"
```

USING C AND C++ METHODS IN NIM



```nim
proc printf(formatstr: cstring){.header: "<stdio.h>", importc: "printf", varargs.}

printf("%d\n", 2620)
printf("Hello world!")
```



```nim
type
  VideoMode {.importcpp: "sf::VideoMode".} = object
  RenderWindowObj {.importcpp: "sf::RenderWindow".} = object
  RenderWindow = ptr RenderWindowObj
  Color {.importcpp: "sf::Color".} = object
  Event {.importcpp: "sf::Event".} = object


proc videoMode(modeWidth, modeHeight: cuint,
                modeBitsPerPixel: cuint = 32): VideoMode
proc newRenderWindow(mode: VideoMode, title: cstring): RenderWindow
proc pollEvent(window: RenderWindow, event: var Event): bool
proc newColor(red, green, blue, alpha: uint8): Color
proc clear(window: RenderWindow, color: Color)
proc display(window: RenderWindow)
```

- The importc pragma provides a means to import a proc or a variable from C.
- Similar to the importc pragma for C, the importcpp pragma can be used to import C++ methods or C++ symbols in general.

SOURCE

- Everything included in the slides was taken from the Nim's official website.
  - https://nim-lang.org

THANK YOU