Nim (formerly Nimrod) is a statically typed, imperative programming language that gives the programmer power without compromises on runtime efficiency.

Nim is efficient, expressive, and elegant.

```
var                     # Declare (and assign) variables,
  letter: char = 'n'    # with or without type annotations
  lang = "N" & "im"
  nLength : int = len(lang)
  boat: float
  truth: bool = false

let            # Use let to declare and bind variables *once*.
  legs = 400   # legs is immutable.
  arms = 2_000 # _ are ignored and are useful for long numbers.
  aboutPi = 3.15

const            # Constants are computed at compile time. This provides
  debug = true   # performance and is useful in compile time expressions.
  compileBadCode = false

when compileBadCode:            # `when` is a compile time `if`
  legs = legs + 1               # This error will never be compiled.
  const input = readline(stdin) # Const values must be known at compile time.

discard 1 > 2 # Note: The compiler will complain if the result of an expression
              # is unused. `discard` bypasses this.

discard """
This can work as a multiline comment.
Or for unparsable, broken code
"""

#
# Data Structures
#

# Tuples

var
  child: tuple[name: string, age: int]   # Tuples have *both* field names
  today: tuple[sun: string, temp: float] # *and* order.

child = (name: "Rudiger", age: 2) # Assign all at once with literal ()
today.sun = "Overcast"            # or individual fields.
today.temp = 70.1

# Sequences

var
  drinks: seq[string]

drinks = @["Water", "Juice", "Chocolate"] # @[V1,..,Vn] is the sequence literal

drinks.add("Milk")
```

```
if "Milk" in drinks:
  echo "We have Milk and ", drinks.len - 1, " other drinks"

let myDrink = drinks[2]

#
# Defining Types
#

# Defining your own types puts the compiler to work for you. It's what makes
# static typing powerful and useful.

type
  Name = string # A type alias gives you a new type that is interchangable
  Age = int     # with the old type but is more descriptive.
  Person = tuple[name: Name, age: Age] # Define data structures too.
  AnotherSyntax = tuple
    fieldOne: string
    secondField: int

var
  john: Person = (name: "John B.", age: 17)
  newage: int = 18 # It would be better to use Age than int

john.age = newage # But still works because int and Age are synonyms

type
  Cash = distinct int    # `distinct` makes a new type incompatible with its
  Desc = distinct string # base type.

var
  money: Cash = 100.Cash # `.Cash` converts the int to our type
  description: Desc  = "Interesting".Desc

when compileBadCode:
  john.age  = money        # Error! age is of type int and money is Cash
  john.name = description  # Compiler says: "No way!"

#
# More Types and Data Structures
#

# Enumerations allow a type to have one of a limited number of values

type
  Color = enum cRed, cBlue, cGreen
  Direction = enum # Alternative formating
    dNorth
    dWest
    dEast
    dSouth
var
  orient = dNorth # `orient` is of type Direction, with the value `dNorth`
```

```
  pixel = cGreen # `pixel` is of type Color, with the value `cGreen`

discard dNorth > dEast # Enums are usually an "ordinal" type

# Subranges specify a limited valid range

type
  DieFaces = range[1..20] # Only an int from 1 to 20 is a valid value
var
  my_roll: DieFaces = 13

when compileBadCode:
  my_roll = 23 # Error!

# Arrays

type
  RollCounter = array[DieFaces, int]  # Array's are fixed length and
  DirNames = array[Direction, string] # indexed by any ordinal type.
  Truths = array[42..44, bool]
var
  counter: RollCounter
  directions: DirNames
  possible: Truths

possible = [false, false, false] # Literal arrays are created with [V1,..,Vn]
possible[42] = true

directions[dNorth] = "Ahh. The Great White North!"
directions[dWest] = "No, don't go there."

my_roll = 13
counter[my_roll] += 1
counter[my_roll] += 1

var anotherArray = ["Default index", "starts at", "0"]

# More data structures are available, including tables, sets, lists, queues,
# and crit bit trees.
# http://nim-lang.org/docs/lib.html#collections-and-algorithms

#
# IO and Control Flow
#

# `case`, `readLine()`

echo "Read any good books lately?"
case readLine(stdin)
of "no", "No":
  echo "Go to your local library."
of "yes", "Yes":
  echo "Carry on, then."
else:
```

```
  echo "That's great; I assume."

# `while`, `if`, `continue`, `break`

import strutils as str # http://nim-lang.org/docs/strutils.html
echo "I'm thinking of a number between 41 and 43. Guess which!"
let number: int = 42
var
  raw_guess: string
  guess: int
while guess != number:
  raw_guess = readLine(stdin)
  if raw_guess == "": continue # Skip this iteration
  guess = str.parseInt(raw_guess)
  if guess == 1001:
    echo("AAAAAAGGG!")
    break
  elif guess > number:
    echo("Nope. Too high.")
  elif guess < number:
    echo(guess, " is too low")
  else:
    echo("Yeeeeeehaw!")

#
# Iteration
#

for i, elem in ["Yes", "No", "Maybe so"]: # Or just `for elem in`
  echo(elem, " is at index: ", i)

for k, v in items(@[(person: "You", power: 100), (person: "Me", power: 9000)]):
  echo v

let myString = """
an <example>
`string` to
play with
""" # Multiline raw string

for line in splitLines(myString):
  echo(line)

for i, c in myString:       # Index and letter. Or `for j in` for just letter
  if i mod 2 == 0: continue # Compact `if` form
  elif c == 'X': break
  else: echo(c)

#
# Procedures
#

type Answer = enum aYes, aNo
```

```
proc ask(question: string): Answer =
  echo(question, " (y/n)")
  while true:
    case readLine(stdin)
    of "y", "Y", "yes", "Yes":
      return Answer.aYes  # Enums can be qualified
    of "n", "N", "no", "No":
      return Answer.aNo
    else: echo("Please be clear: yes or no")

proc addSugar(amount: int = 2) = # Default amount is 2, returns nothing
  assert(amount > 0 and amount < 9000, "Crazy Sugar")
  for a in 1..amount:
    echo(a, " sugar...")

case ask("Would you like sugar in your tea?")
of aYes:
  addSugar(3)
of aNo:
  echo "Oh do take a little!"
  addSugar()
# No need for an `else` here. Only `yes` and `no` are possible.

#
# FFI
#

# Because Nim compiles to C, FFI is easy:

proc strcmp(a, b: cstring): cint {.importc: "strcmp", nodecl.}

let cmp = strcmp("C?", "Easy!")
```

Additionally, Nim separates itself from its peers with metaprogramming, performance, and compile-time features.

Further Reading

- Home Page
- Download
- Community
- FAQ
- Documentation
- Manual
- Standard Library
- Rosetta Code