



## Basic Syntax and Data Types

### Fundamental Syntax

- # - Single-line comment.
- ## - Multi-line comment.
- ;- Statement separator (usually optional).

**Example:**

```
# This is a single-line comment
## This is a
## multi-line comment
let x = 5; echo x # Statement separator
used here
```

- let - Immutable variable.
- var - Mutable variable.
- const - Compile-time constant.

**Example:**

```
let immutable = 10
var mutable = 20
mutable = 30 # OK
const compileTime = 40
# compileTime = 50 # Error: cannot
assign to 'compileTime'
```

- echo - Print to standard output.

**Example:**

```
echo "Hello, Nim!"
echo 1, 2, 3 # Prints: 1 2 3
```

### Basic Data Types

<code>int</code>	Integer (platform-dependent size). Example: <code>let x: int = 10</code>
<code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>	Signed integers of specific sizes. Example: <code>let y: int32 = -1000</code>
<code>uint</code>	Unsigned integer (platform-dependent size). Example: <code>let z: uint = 100</code>
<code>uint8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code>	Unsigned integers of specific sizes. Example: <code>let a: uint16 = 65535</code>
<code>float</code> , <code>float32</code> , <code>float64</code>	Floating-point numbers. Example: <code>let b: float = 3.14</code>
<code>bool</code>	Boolean (true or false). Example: <code>let c: bool = true</code>
<code>char</code>	Single character. Example: <code>let d: char = 'A'</code>
<code>string</code>	Sequence of characters. Example: <code>let e: string = "Hello"</code>

### String Literals

- " - Regular string literal.
- r" - Raw string literal (no escape sequences).
- """ - Long string literal (can span multiple lines).

**Example:**

```
let normal = "Hello\nWorld"
let raw = r"Hello\nWorld"
let long = """
This is a
long string.
"""
echo normal # Prints: Hello\nWorld
echo raw # Prints: Hello\nWorld
echo long # Prints: This is a\nlong
string.
```

## Control Flow and Procedures

### Conditional Statements

**if** - Basic conditional statement.  
**elif** - Else if.  
**else** - Else.

#### Example:

```
let x = 10
if x > 0:
  echo "Positive"
elif x < 0:
  echo "Negative"
else:
  echo "Zero"
```

**case** - Switch statement.

#### Example:

```
let day = "Monday"
case day
of "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday":
  echo "Weekday"
of "Saturday", "Sunday":
  echo "Weekend"
else:
  echo "Invalid day"
```

### Loops

**for** loop  
Iterating over a range or collection.

```
for i in 0..5:
  echo i # Prints 0 to 5
```

```
let arr = [1, 2, 3]
for item in arr:
  echo item # Prints 1,
2, 3
```

**while** loop  
Looping while a condition is true.

```
var i = 0
while i < 5:
  echo i
  i += 1
```

**break**  
statement  
Exits the current loop.

```
for i in 0..10:
  if i > 5:
    break
  echo i # Prints 0 to 5
```

**continue**  
statement  
Skips the current iteration and continues with the next.

```
for i in 0..5:
  if i mod 2 == 0:
    continue
  echo i # Prints 1, 3,
5
```

### Procedures

**proc** - Define a procedure.

#### Syntax:

```
proc add(x, y: int): int =
  return x + y
```

```
let result = add(5, 3)
echo result # Prints 8
```

Parameters can have default values:

```
proc greet(name: string = "World") =
  echo "Hello, " & name
```

```
greet() # Prints: Hello, World
greet("Nim") # Prints: Hello, Nim
```

Discarding return values:

```
proc sayHello(): string =
  return "Hello"
```

```
discard sayHello()
```

## Collections and Data Structures

### Arrays and Sequences

**array** - Fixed-size collection.  
**seq** - Dynamic array (sequence).

#### Example:

```
let arr: array[3, int] = [1, 2, 3]
var seq1: seq[int] = @[4, 5, 6] # @[]
creates a sequence
seq1.add(7)
echo seq1 # Prints: @[4, 5, 6, 7]
```

#### Array access:

```
let value = arr[0] # Access the first
element (index 0)
echo value # Prints: 1
```

#### Sequence length:

```
echo seq1.len # Prints: 4
```

## Tuples and Objects

**tuple** Collection of named fields with different types.

```
type Person = tuple[name: string,
age: int]
let person1: Person = (name:
"Alice", age: 30)
echo person1.name # Prints: Alice
echo person1.age # Prints: 30
```

**object** User-defined type with fields and methods (similar to classes in other languages).

```
type
Rectangle = object
width: float
height: float

proc area(r: Rectangle): float =
return r.width * r.height

var rect: Rectangle
rect.width = 5.0
rect.height = 3.0
echo area(rect) # Prints: 15.0
```

## Sets and Dictionaries

**set** - Collection of unique elements.  
**Table** - Hash table (dictionary).

### Example:

```
import tables
import sets

var mySet: set[int] = {1, 2, 3}
mySet.incl(4)
echo mySet # Prints: {1, 2, 3, 4}

var myTable = initTable[string, int]()
myTable["Alice"] = 30
myTable["Bob"] = 25
echo myTable["Alice"] # Prints: 30
```

## Advanced Features

### Generics

Generic procedures and types allow you to write code that works with multiple types.

### Example:

```
proc identity[T](x: T): T =
return x

let intValue = identity[int](5) #
intValue is 5
let stringValue = identity[string]
("Hello") # stringValue is "Hello"

echo intValue
echo stringValue
```

**static** Evaluate code at compile time. Useful for generating code or performing calculations at compile time.

```
static:
  let compileTimeValue = 2 * 2
  echo "Compile time value: ",
  compileTimeValue # Prints during
  compilation
```

**template** Code generation mechanism. Templates are expanded at compile time.

```
template twice(x: expr): expr =
  x * 2

let result = twice(5) # expands to
5 * 2
echo result          # Prints: 10
```

**macro** More powerful than templates. Macros can manipulate the abstract syntax tree (AST) of the code.

```
import macros
macro assert(cond: expr, msg:
string = "Assertion failed"):
  result = quote do:
    if not `cond`:
      raise newException(Defect,
`msg`)

assert(1 == 1) # OK
# assert(1 == 2, "Custom message")
# Raises an exception at runtime
```

**try**, **except**, **finally** - Exception handling.

**Example:**

```
try:
  let result = 10 div 0 # Raises an
  exception
  echo result
except DivByZeroError:
  echo "Division by zero error!"
finally:
  echo "This will always be executed."
```

**raise** - Raise an exception.

**Example:**

```
raise newException(ValueError, "Invalid
value!")
```