

---

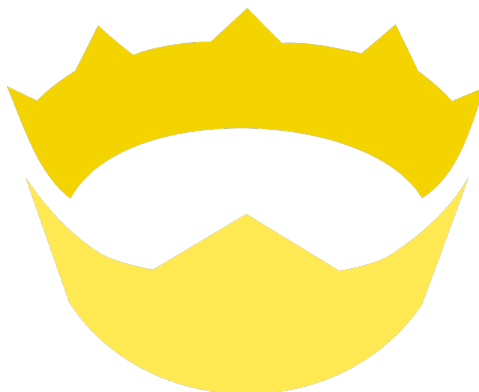
# Podstawy programowania w języku Nim

## podręcznik dla początkujących

---

Namysław Tatarynowicz

Przygotowano 5 lipca 2023



## Spis treści

<b>1</b>	<b>Wprowadzenie do języka Nim</b>	<b>2</b>
1.1	Wstęp . . . . .	2
1.2	Poznajmy Nima . . . . .	2
1.3	Przygotowanie środowiska . . . . .	4
1.4	Kompilacja . . . . .	4
1.5	Pierwszy program . . . . .	6
<b>2</b>	<b>Zmienne, stałe i typy danych</b>	<b>10</b>
2.1	Zmienne i stałe . . . . .	10
2.2	Typy danych . . . . .	12
<b>3</b>	<b>Praca z tekstami</b>	<b>14</b>
<b>4</b>	<b>Instrukcje warunkowe</b>	<b>18</b>
<b>5</b>	<b>Funkcje</b>	<b>23</b>
<b>6</b>	<b>Moduły</b>	<b>25</b>
6.1	Instalacja i import modułów . . . . .	25
6.2	Własne moduły . . . . .	27
<b>7</b>	<b>Sekwencje i moduł sequtils</b>	<b>29</b>
7.1	Sekwencje . . . . .	29
7.2	Moduł sequtils . . . . .	30
<b>8</b>	<b>Pętle</b>	<b>35</b>
8.1	Pętla while . . . . .	35
8.2	Pętla for . . . . .	37
<b>9</b>	<b>Tablice</b>	<b>39</b>
<b>10</b>	<b>Tuple</b>	<b>43</b>
<b>11</b>	<b>Praca z plikami tekstowymi</b>	<b>45</b>
11.1	Odczyt danych z pliku . . . . .	45
11.2	Zapis danych do pliku . . . . .	47

# 1 Wprowadzenie do języka Nim

## 1.1 Wstęp

### Uwaga

*Ten podręcznik nie jest oficjalnym dokumentem i nie jest powiązany z [nim-lang.org](https://nim-lang.org).*

Tworząc ten dokument, przyjąłem sobie za cel przedstawienie podstaw programowania w Nim. Jest to również sposób na moją naukę tego języka. Zawartość dokumentu będzie ewoluowała wraz ze wzrostem mojej świadomości programowania w Nim.

Niniejszy podręcznik jest w bardzo wczesnej fazie tworzenia. Posiada wiele elementów do poprawy, o których wiem, ale też mnóstwo, z których jeszcze nie zdaję sobie sprawy. Gdy zauważysz merytoryczny błąd lub masz sugestię, proszę o kontakt na adres mailowy: [n.tatarynowicz@gmail.com](mailto:n.tatarynowicz@gmail.com).

Opracowując ten dokument, korzystam z następującego oprogramowania:

system operacyjny: antiX 22  
język : Nim 1.6.12  
kompilator : glibc 2.28  
edytor kodu : Geany 1.33

## 1.2 Poznajmy Nima

Nim jest językiem programowania o składni zbliżonej do języka Python, ale kompilowanym. Dzięki temu umożliwia tworzenie programów o podobnej wydajności jak programy napisane w języku C. Nim do kompilacji kodu wykorzystuje jedną z dostępnych bibliotek języka C/C++ (domyślnie C). Zatem, podczas kompilacji programu, najpierw powstaje plik C i dopiero ten plik jest kompilowany. Ostatecznie Nim daje możliwość kompilacji do C, C++ lub JavaScript.

Podobnie, jak w języku Python, w Nim także możemy korzystać z gotowych modułów, które importujemy tak samo w obu językach. Trzeba zauważyć, że Nim nie posiada tak dużej ich liczby, jak Python.

Najnowszą wersję Nima znajdziemy na stronie <https://nim-lang.org/>. W momencie tworzenia tego dokumentu, na stronie twórców, Nim jest dostępny w wersji 1.6.12.

Znajduje się tam również środowisko, dzięki któremu będziemy mogli skompilować i uruchomić nasz kod on-line oraz wygenerować odnośnik i go udostępnić: <https://play.nim-lang.org/>.

Język Nim jest wolnym oprogramowaniem — jest rozprowadzany na licencji MIT.

Jeśli napotkamy na jakiś programistyczny problem, to dzięki społeczności Nima możemy uzyskać pomoc w jego rozwiązaniu. Poniżej umieszczam listę miejsc, które możemy odwiedzić. W momencie przygotowywania tego podręcznika nie spotkałem polskojęzycznych kanałów społecznościowych przeznaczonych użytkownikom Nima.

**blog:**

<https://nim-lang.org/blog.html>

**forum:**

<https://forum.nim-lang.org/>

**IRC:**

[irc.libera.chat](#), #nim

**Discord:**

<https://discord.gg/nim>

**Matrix:**

<https://matrix.to/#/#nim-lang:matrix.org>

**Telegram:**

[https://t.me/nim\\_lang](https://t.me/nim_lang)

**Twitter:**

[https://twitter.com/nim\\_lang](https://twitter.com/nim_lang)

**Reddit:**

<https://www.reddit.com/r/nim>

**StackOverflow:**

<https://stackoverflow.com/questions/tagged/nim-lang>

**Wiki:**

<https://github.com/nim-lang/Nim/wiki>

### 1.3 Przygotowanie środowiska

Abyśmy mogli zacząć pisać i kompilować programy na naszym komputerze, będziemy potrzebowali:

- pakietu Nim
- kompilatora języka C/C++
- dowolnego/ulubionego edytora kodu

Nim nie dostarcza zintegrowanego środowiska programistycznego, dlatego wszystkie powyższe elementy będziemy musieli pobrać, zainstalować i skonfigurować samodzielnie.

W przypadku systemu operacyjnego Windows ze strony projektu należy pobrać archiwum zip Nima w wersji odpowiedniej do posiadanej architektury komputera. Analogicznie należy postąpić ze znajdującym się na stronie archiwum kompilatora języka C (mingw).

Chcąc zainstalować Nima na jednej z dystrybucji GNU/Linux można to zrobić na dwa sposoby. Możemy zainstalować pakiet Nim z repozytorium danej dystrybucji lub pobrać go ze strony projektu. W obu przypadkach dodatkowo będzie potrzebny kompilator C/C++. Warto wówczas zainstalować metapakiet o nazwie **build-essentials** zawierający niezbędne biblioteki wykorzystywane do pracy z językami C/C++.

### 1.4 Kompilacja

Najszybszym sposobem skompilowania przygotowanego pliku jest wykonanie polecenia:

```
nim c nazwa_pliku.nim
```

Po poprawnym skompilowaniu otrzymamy wykonywalny plik o takiej samej nazwie.

Ogólna składnia kompilacji programu jest następująca:

```
nim polecenie [opcje] [nazwa_pliku] [argumenty]
```

**polecenie** może przyjmować następujące wartości:

<b>compile, c</b>	kompiluje projekt z wykorzystaniem domyślnego generatora kodu C
<b>r</b>	kompiluje do \$nimcache/projname i uruchamia z argumentami domyślnie używając kodu C
<b>doc</b>	generuje dokumentację dla pliku wejściowego

opcje mogą przyjmować następujące wartości:

<code>-p, --path:PATH</code>	dodaj ścieżkę do ścieżek wyszukiwania
<code>-d, --define:SYMBOL(:VAL)</code>	zdefiniuj symbol warunkowy
<code>-u, --undef:SYMBOL</code>	usuń definicję symbolu warunkowego
<code>-f, --forceBuild:on off</code>	wymuś przebudowę wszystkich modułów
<code>--stackTrace:on off</code>	włącz/wyłącz ślad stosu
<code>--threads:on off</code>	włącz/wyłącz obsługę wielowątkowości
<code>-x, --checks:on off</code>	włącz/wyłącz wszystkie kontrole środowiska uruchomieniowego
<code>-a, --assertions:on off</code>	włącz/wyłącz asercje
<code>--opt:none speed size</code>	nie optymalizuj, optymalizuj pod względem szybkości lub czasu
<code>--debugger:native</code>	użyj natywnego debugera (gdb)
<code>--app:console gui lib staticlib</code>	wygeneruj aplikację konsolową, graficzną, DLL lub bibliotekę statyczną
<code>-r, --run</code>	uruchom skompilowany program z podanymi argumentami
<code>--eval:cmd</code>	bezpośrednia ewaluacja kodu nim; np.: <code>nim --eval:"echo 1"</code>
<code>--fullhelp</code>	pokaż pełny podręcznik pomocy
<code>-h, --help</code>	pokaż podręcznik pomocy
<code>-v, --version</code>	pokaż szczegóły dot. zainstalowanej wersji

Przykładowa konstrukcja polecenia do skompilowania kodu zawartego w pliku `main.nim`:

```
nim c -r --opt:speed main.nim
```

Dzięki czemu kod (zapisany w pliku `main.nim`) najpierw zostanie skompilowany ( `c` ) z optymalizacją pod względem szybkości działania ( `--opt:speed` ), a następnie uruchomiony ( `-r` ).

## 1.5 Pierwszy program

Zazwyczaj podczas nauki programowania nowego języka, jako pierwszy, pisze się program wyświetlający napis `Hello world`. Więc tradycji niech stanie się zadość.

### Ważne

Wszystkie pliki źródłowe Nima muszą być kodowane z wykorzystaniem UTF-8 (lub jego podzbiorze ASCII). Żadne inne kodowanie nie jest wspierane.

Do wyświetlania tekstu na ekranie służy polecenie `echo`. Konsekwencją jego działania, poza wyświetlaniem tekstu, jest przejście kursora do nowej linii.

```
echo "Hello world"
```

```
Hello world
```

```
-
```

Istnieją różne sposoby wyświetlania tekstu w konsoli:

```
echo "Hello world"
echo("Hello world")
"Hello world".echo()
"Hello world".echo
"Hello ".echo("world")
"Hello".echo " world"
```

Wszystkie powyższe przykłady wyświetlą napis `Hello world`, po czym kursor przejdzie do nowej linii.

Jeśli chcemy, aby po wyświetleniu tekstu kursor pozostał w tej samej linii, skorzystamy z `stdout.write`. Możemy i tu wymusić przejście do nowej linii używając `\n`

```
stdout.write "Hello world. "
stdout.write "Bye!\n"
```

```
Hello world. Bye!
```

```
-
```

Można oczywiście wyświetlać liczby lub wyniki operacji arytmetycznych. Liczby zmiennoprzecinkowe mają oddzielną część całkowitą od dziesiętnej

kropką.

```
echo 20
echo 3.14
echo 5+5
echo 6-2
echo 2*4
echo 15/2
```

```
20
3.14
10
4
8
7.5
-
```

Pisząc programy, możemy używać następujących operatorów arytmetycznych:

Operator	Nazwa
+	dodawanie
-	odejmowanie
*	mnożenie
/	dzielenie
div	dzielenie bez reszty
mod	reszta z dzielenia

W kodzie źródłowym można stosować komentarze. W Nim mamy do dyspozycji ich dwa rodzaje — jednolinijkowe, zaczynające się symbolem hash `#` i wielolinijkowe — tekst zapisany pomiędzy `#[` a `]#` lub pomiędzy `discard ""` a `""`. Poniższy przykład przedstawia wyżej opisane sposoby.



```
# To jest komentarz jednolinijkowy

echo "Hej!" # można też tak komentować

#[
To jest
komentarz
wielolinijkowy
]#

discard ""
To także jest
komentarz
wielolinijkowy
""
```

Po kompilacji i uruchomieniu pliku zostanie wyświetlony tylko napis "Hej!".

Edytory kodu zwykle posiadają własną opcję wstawiania komentarzy, np. generują znaki komentowania jednolinijkowego, ale dla zaznaczonej części kodu. Taki sposób wydaje się najbardziej wygodny i szybki.

Podobnie, jak w innych językach programowania, także i w Nim można łączyć teksty (ciągi znaków). Takie łączenie nazywa się konkatencją.

Gdy mamy przygotowane już napisy do połączenia, wówczas możemy użyć znaku `&`.

```
echo "Ala " & "ma " & "kota."
```

```
Ala ma kota.
```

Bezpośrednio można łączyć ze sobą tylko ciągi znaków. Jeśli mamy do złączenia ciąg znaków i liczbę, wówczas trzeba tę liczbę rzutować na ciąg znaków. Do rzutowania danego typu danych na ciąg znaków użyjemy symbolu `$`, który wstawimy przed liczbą.

```
echo "Ala " & "ma " & $20 & " lat."
```

```
Ala ma 20 lat.
```

Można też to zrobić trochę szybciej, używając znaku `,`, który automatycznie zamienia wszystkie argumenty na ciągi znaków.

```
echo "Ala ", "ma ", 20, " lat."
```

```
Ala ma 20 lat.
```

```
-
```

Na tym etapie nauki mogę zasugerować przyjęcie jednego, wygodnego sposobu zarówno wyświetlania tekstów, tworzenia komentarzy, zwłaszcza wielolinijkowych, jak i łączenia tekstów. Pomoże to w utrzymaniu porządku w naszym kodzie i sprawi, że będzie on bardziej czytelny.

## 2 Zmienne, stałe i typy danych

### 2.1 Zmienne i stałe

Zmienne w programowaniu często porównuje się do pojemników, które przechowują różne wartości. Tymi wartościami może być pojedynczy znak, ciąg znaków, różnego rodzaju liczby, a nawet wartości logiczne — prawda, czy fałsz. Odwołując się do tych pojemników, możemy użyć wartości w nich przechowywanych.

W Nim przed użyciem zmiennej trzeba ją wcześniej zadeklarować. Do deklaracji zmiennej służy `var`. Spójrzmy na przykład:

```
var name = "Adam"

echo "Cześć! Mam na imię ", name, "."
```

```
Cześć! Mam na imię Adam.
-
```

Deklaracja zmiennej polega na określeniu nazwy, pod jaką będzie ona przechowywała wartości oraz typu tych danych. Nim „potrafi” rozpoznać, jakiego typu jest zmienna, którą przechowujemy. Jednak, gdy chcemy zadeklarować zmienną bez podawania jej wartości, musimy sami określić jej typ:

```
var age: int
```

Właśnie zadeklarowaliśmy zmienną o nazwie `age`, która będzie przechowywała liczby całkowite (`int`).

Nic nie stoi na przeszkodzie, żeby oprócz przypisania wartości do zmiennej, określić również jej typ:

```
var age: int = 20
```

Po deklaracji zmiennej można zmieniać jej wartość, ale pamiętajmy, że nowe wartości muszą być tego samego typu.

```
var age: int = 20
echo age
age = 17
echo age
```

```
20
17
-
```

Jeśli deklarujemy kilka zmiennych, możemy uprościć zapis, stosując wcięcia:

```
var
  x = 10
  y = 3.5
  z = "Hello!"
```

Oprócz zmiennych w Nim możemy używać stałych. Służą one do przechowywania wartości, których, gdy zostaną już określone, nie będzie możliwości dokonania ich zmiany.

```
const pi = 3.14159
```

Proszę zwrócić uwagę, że **każda stała zadeklarowana w ten sposób musi mieć ustaloną wartość już podczas kompilacji programu.**

Podobnie, jak w przypadku zmiennych, dla deklaracji stałych także możemy przygotować deklarację blokową:

```
const
  pi = 3.14159
  c = 300_000_000
  text = "Hello world"
```

W powyższym przykładzie został użyty trochę inny, niż do tej pory sposób zapis liczb: 300\_000\_000. Ułatwia on poprawne wpisywanie bardzo dużych (lub bardzo małych) wartości, ale nie zmienia on interpretacji tej liczby. Przykład:

```
echo 300_000_000
```

```
300000000
-
```

W Nim można również deklarować stałe, które nie muszą mieć ustalonych wartości podczas kompilacji. Deklarujemy je, używając `let`:

```

stdout.write("Podaj swoje imię: ")
let name = stdin.readLine()

echo name

```

W powyższym przykładzie użytkownik zostanie poproszony o podanie imienia, a następnie to imię wyświetli się na ekranie.

W Nim możemy tworzyć zmienne lub stałe, których nazwy składają się z kilku słów, pomiędzy którymi może być wstawiona spacja. Musimy wówczas nazwę zmiennej zamknąć pomiędzy ``` a ```. Co więcej, odwołując się do nazw zmiennych lub stałych Nim nie uwzględnia wielkości znaków poza pierwszym znakiem. Proszę zwrócić uwagę na poniższy przykład.

```

let `my own constant` = 12

echo `my own constant`
echo myownconstant
echo myOwnConstant

```

```

12
12
12
-

```

Gdybyśmy jednak próbowali odwołać się do stałej o nazwie `MyOwnConstant` otrzymalibyśmy błąd: `undeclared identifier: 'MyOwnConstant'`.

## 2.2 Typy danych

W Nim możemy korzystać z następujących typów danych:

```

var c: char = 'y'           # pojedynczy znak
var name: string = "Adam"   # ciąg znaków
var age: int = 10            # liczba całkowita
var pi: float = 3.14159     # liczba zmiennoprzecinkowa
var b: bool = true          # wartość logiczna

```

Podczas deklaracji zmiennej lub stałej, która będzie przechowywała pojedynczy znak, zamiast cudzysłowów używamy apostrofów.

Żeby sprawdzić typ danej zmiennej lub stałej korzystamy z atrybutu `type`. Oto typy danych dla zmiennych z powyższego przykładu:

```
echo c.type
echo name.type
echo age.type
echo pi.type
echo b.type
```

```
char
string
int
float
bool
-
```

W przypadku liczb całkowitych możemy dodatkowo określić zakres. Mamy do dyspozycji: `int64`, `int32`, `int16`, `int8`. Zatem możemy ustalić zakresy liczb całkowitych, kolejno: 64 bitowe, 32 bitowe, 16 bitowe i 8 bitowe.

Analogicznie mamy do dyspozycji liczby całkowite bez znaku (liczby dodatnie i 0): `uint64`, `uint32`, `uint16`, `uint8`.

Możemy sprawdzić, jak dużo miejsca w pamięci zajmuje stała lub zmienna. Wykorzystamy do tego operator `sizeof`. Wartość ta jest podawana w bajtach.

```
var a: uint64 = 10
var b: uint32 = 10
var c: uint16 = 10
var d: uint8 = 10

echo a.sizeof
echo b.sizeof
echo c.sizeof
echo d.sizeof
```

```
8
4
2
1
-
```

Dobranie odpowiedniego typu do przechowywania danych pozwala zoptymalizować ilość pamięci potrzebną do działania naszego programu.

### 3 Praca z tekstami

W języku Nim możemy odwoływać się do poszczególnych znaków tekstu lub do pozycji (tzw. numer indeksu), na których one się znajdują. Warto zapamiętać, że **numeracja indeksów rozpoczyna się od 0**. Poniżej umieszczam przykładowe zestawienie.

```
let text = "Hello world"

echo text[0]           # pierwszy znak
echo text[1..8]        # znaki od indeksu 1 do 8
echo text[^1]          # ostatni znak
echo text[^2]          # przedostatni znak
echo text[^3..^1]      # ostatnie trzy znaki
echo text.low           # pierwszy indeks
echo text.high          # ostatni indeks
echo text[text.low]    # znak spod pierwszego indeksu
echo text[text.high]   # znak spod ostatniego indeksu
```

```
H
ello wor
d
l
rld
0
10
H
d
-
```

Używając funkcji `add()`, można w łatwy sposób dołączać nowe ciągi znaków do już istniejących tzw. metodą inline, czyli bezpośrednio nadpisywać wartości zmiennej.

```
var text = "Lorem"
text.add(" ipsum")

echo text
```

```
Lorem ipsum
-
```

Do pomiaru długości ciągu znaków służy funkcja `len()`. Można z niej skorzystać na różne sposoby.

```
var text = "Lorem"

echo len text
echo len(text)
echo text.len
echo text.len()
```

Wszystkie powyższe przykłady wyświetlą długość tekstu zapisanego pod zmienną `text`, czyli 5.

Możemy zmienić długość wartości zmiennej tekstowej. W tym celu użyjemy `setlen()`. Zmieńmy długość ciągu na 3 znaki.

```
var text = "Lorem ipsum"
text.setlen(3)

echo text
```

```
Lor
```

```
-
```

Istnieje również możliwość tworzenia wielolinijkowych ciągów znaków, które zachowują także znaki nowej linii. Należy wówczas użyć potrójnych cudzysłówów — zamknąć tekst pomiędzy `"""` a `"""`.

```
let text = """
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt...
"""

echo text
```

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt...
```

```
-
```

Wyświetlając teksty w konsoli, możemy używać znaków specjalnych, tych poprzedzonych symbolem backslash `\`. Takim znakiem specjalnym jest zna-



ny już nam z początku podręcznika znak nowej linii `\n`. Innym znakiem specjalnym jest `\t` wstawiający znak tabulacji. Poniższy przykład pokazuje, jak w konsoli będą wyglądały słowa oddzielone takim właśnie znakiem.

```
echo "pon.\twtó.\tśr.\tczw.\tpią."
```

```
pon.    wtó.    śr.    czw.    pią.  
-
```

Symbol backslash oprócz tego, że czyni ze zwykłych znaków znaki specjalne, może także zamieniać znaki specjalne na zwykłe. Np. gdybyśmy chcieli wyświetlić tekst, który zawiera w sobie cudzysłów, wówczas musimy poprzedzić ten cudzysłów znakiem backslash. Oto przykład:

```
echo "Oto cytat: \"Carpe diem\"."
```

```
Oto cytat: "Carpe diem".  
-
```

Wcześniej poznaliśmy jak łączyć ze sobą ciągi znaków oraz ciągi z liczbami. Jednak gdy mamy do złączenia więcej elementów, np. tekst i zmienne, wygodniej jest użyć formatowania z modułu `strformat`. Jednocześnie pierwszy raz zaimportujemy moduł do naszego programu.

```
import std/strformat  
  
let  
  name = "Adam"  
  age = 19  
  hobby = "programowanie w Nim"  
  
echo &"{name} ma {age} lat i jego hobby to {hobby}."
```

```
Adam ma 19 lat i jego hobby to programowanie w Nim.  
-
```

Zamiast znaku `&` w powyższym przykładzie, można użyć `fmt`, co przyniesie taki sam efekt. Różnica w tych dwóch sposobach formatowania tekstów jest taka, że `fmt` nie interpretuje znaków specjalnych i wyświetla je, jak surowy tekst. Oto przykład:

```
import std/strformat  
  
echo &"Ala ma\tkota."  
echo fmt"Ala ma\tkota."
```

```
Ala ma    kota.  
Ala ma\tkota.  
-
```

## 4 Instrukcje warunkowe

Instrukcje warunkowe pozwalają komputerowi na podejmowanie decyzji. Prześledźmy poniższy pseudokod:

```
if warunek:
    jeśli warunek jest spełniony, realizowany jest ten blok
elif warunek:
    jeśli powyższy warunek nie jest spełniony,
    sprawdzany jest ten warunek i jeśli jest spełniony
    realizowany jest ten blok
else:
    jeśli żaden z powyższych warunków nie jest spełniony,
    realizowany jest ten blok
```

W powyższym przykładzie, po dwukropkach, zostały użyte tzw. wcięcia. W Nim, podobnie jak w Pythonie, wcięcia służą do wydzielania części kodu, który ma zostać wykonany. Wcięciem może być pojedyncza spacja lub jej wielokrotność. Natomiast **wcięciem nie może być tabulacja**. Ważne, żeby liczba spacji tworzących wcięcie była taka sama, za każdym razem, gdy robimy je na jednym poziomie. Wygodnie jest, gdy edytor, którego używasz, potrafi wstawiać spacje, gdy naciśniesz klawisz tabulatora. W moim edytorze kodu (Geany), właśnie tak jest i mam ustawione wcięcia na dwa znaki spacji. W przykładach zawartych w tym podręczniku będę stosował także takie wcięcia. Dokumentacja Nima zaleca stosowanie dokładnie takiej liczby spacji tworzących wcięcie.

Gdy program ma sprawdzić jeden warunek, używamy `if`. Jeśli program ma do sprawdzenia kilka warunków, użyjemy wtedy konstrukcji `if` i `elif`. W momencie, gdy jeden z nich jest spełniony, pozostałe nie są wówczas sprawdzane. Z `else` korzystamy wówczas, gdy program ma zrealizować jakiś kod, gdy żaden z powyższych warunków nie został spełniony.

Oto przykład programu określającego czy osoba posiadająca dany wiek jest pełnoletnia, czy nie:

```
let age = 17
if age < 18:
    echo "Nie jesteś pełnoletni."
else:
    echo "Jesteś pełnoletni."
```

```
Nie jesteś pełnoletni.
```

```
-
```

Kolejny przykład. Tym razem wykorzystamy `if`, `elif` i `else`:

```
let satellite = "Fobos"

if satellite == "Deimos":
    echo "Deimos jest mniejszym z dwóch księżyców Marsa."
elif satellite == "Fobos":
    echo "Fobos jest większym z dwóch księżyców Marsa."
else:
    echo "Mars nie posiada takiego księżycyca..."
```

```
Fobos jest większym z dwóch księżyców Marsa.
-
```

W powyższym przykładzie został użyty podwójny znak równości `==` będący operatorem porównania. Korzystamy z niego wówczas, gdy chcemy sprawdzić, czy jedna wartość jest równa drugiej. Z kolei pojedynczy znak równości `=` jest tzw. operatorem przypisania. Korzystamy z niego, gdy np. do zmiennej chcemy przypisać wartość.

Poniżej tabela zawierająca zestawienie operatorów porównania:

Operator	Opis
<code>==</code>	równa się
<code>!=</code>	nie równa się
<code>&lt;</code>	mniejsze niż
<code>&gt;</code>	większe niż
<code>&lt;=</code>	mniejsze lub równe
<code>&gt;=</code>	większe lub równe

Gdy np. do sprawdzenia mamy więcej niż jeden warunek, wówczas możemy skorzystać z operatorów logicznych. Operatory logiczne służą do testowania prawdziwości wyrażenia składającego się z jednej lub więcej wartości logicznych. Mamy do dyspozycji następujące operatory logiczne: `and` (i), `or` (lub), `xor` (alternatywa wykluczająca), `not` (nie jest).

- logiczne `and` zwraca `true` tylko wówczas, gdy wszystkie elementy biorące udział w teście zwracają `true`,
- logiczne `or` zwraca `true` jeśli chociaż jeden element biorący udział w teście zwraca `true`,
- logiczne `xor` zwraca `true` jeśli jeden element zwraca `true` a pozostałym nie,

- logiczne `not` neguje prawdziwość swojego elementu zmieniając `true` na `false` lub odwrotnie.

Przykład wykorzystania operatora logicznego `and` — kalkulator emerytalny:

```
let
  sex = "mężczyzna"
  age = 65

if sex == "mężczyzna" and age >= 65:
  echo "Brawo! możesz iść na emeryturę."
elif sex == "kobieta" and age >= 60:
  echo "Brawo! Możesz iść na emeryturę."
else:
  echo "Niestety, nie możesz jeszcze iść na emeryturę."
```

Brawo! Możesz iść na emeryturę.

-

Jeśli mamy więcej warunków do sprawdzenia można użyć wygodniejszego sposobu — instrukcji `case` i `of`:

```
let pizza = "clasic"

case pizza:
  of "margharita":
    echo "sos, ser, oregano"
  of "clasic":
    echo "sos, ser, salami, pieczarki, szynka, oregano"
  of "romana":
    echo "sos, ser, kurczak, cebula, groszek, oregano"
```

sos, ser, salami, pieczarki, szynka, oregano

-

W powyższym przykładzie po `case pizza:` został wydzielony blok tekstu dla wszystkich przypadków opisanych przez instrukcje `of`. Nie ma konieczności tworzenia takiego bloku. W następnych przykładach nie będę już go stosował.

Kolejny przykład przedstawia możliwość użycia przedziałów liczbowych dla instrukcji `case` i `of`. Sprawdźmy, czy liczba całkowita jest dodatnia, ujemna, czy równa zero:

```
let number = -2

case number
of low(int).. -1:
  echo "Liczba ujemna"
of 0:
  echo "Zero"
of 1..high(int):
  echo "Liczba dodatnia"
```

Liczba ujemna

-

Poniższy kod przedstawia sposób zastąpienia konstrukcji `if`, `elif` z operatorem logicznym `or` instrukcją `case`:

```
let num = 5

# Wersja z if
if num == 2 or num == 3 or num == 5:
  echo "To jest liczba pierwsza."
elif num == 4 or num == 6 or num == 8:
  echo "To jest liczba złożona."
elif num == 0 or num == 1:
  echo "To nie jest ani liczba pierwsza ani liczba złożona."
else:
  echo "Nieprawidłowa wartość."

# Wersja z case
case num
of 2, 3, 5:
  echo "To jest liczba pierwsza."
of 4, 6, 8:
  echo "To jest liczba złożona."
of 0, 1:
  echo "To nie jest ani liczba pierwsza ani liczba złożona."
else:
  echo "Nieprawidłowa wartość."
```

Wynikiem obu wersji programów będzie `To jest liczba pierwsza.`

Do dyspozycji mamy różne sposoby wykorzystania instrukcji warunkowych. Tylko od nas zależy, która konstrukcja będzie wygodniejsza do realizacji naszego zadania.

## 5 Funkcje

Funkcje są wydzielonymi częściami kodu i odgrywają rolę podprogramów. Do funkcji możemy przekazywać argumenty, a ona może, choć nie musi, zwracać wartość.

Funkcje pozwalają na podział programu na logiczne części, dzięki czemu możemy wygodniej zarządzać kodem źródłowym.

Do zdefiniowania funkcji użyjemy polecenia `proc`, a następnie podamy nazwę funkcji zakończoną nawiasami okrągłymi (w następnych przykładach nawiasy te się nam przydadzą). Na końcu umieszczamy znak `=`, który otwierać będzie blok kodu do wykonania. Funkcję możemy wywołać, podając jej nazwę wraz z nawiasami.

Na początek stwórzmy prostą funkcję, która będzie wyświetlała napis `Hello world!`. Spójrz, proszę na poniższy przykład:

```
proc hello() =          # definicja
  echo "Hello world!" # funkcji

hello()                 # wywołanie funkcji
```

```
Hello world!
```

```
-
```

Stwórzmy teraz funkcję, do której prześlemy argument. W nawiasach po nazwie funkcji wpiszemy nazwę zmiennej, pod którą będzie przechowywana wartość wewnątrz funkcji. Musimy również podać, jakiego typu planujemy przekazać wartość. W naszym przykładzie będzie to `string`:

```
proc hi(name: string) =
  echo "Witaj ", name

hi("Adam")
```

```
Witaj Adam
```

```
-
```

Oba powyższe przykłady przedstawiały funkcje, które nie zwracały żadnych wartości — służyły tylko do wyświetlania napisu. Teraz stworzymy funkcję, która taką wartość będzie zwracała, dzięki czemu będziemy ją mogli przypisać do zmiennej:



```
proc sum(x: int, y: int): int =  
  return x + y  
  
var result = sum(3, 4)  
  
echo result
```

```
7  
-
```

Do powyższej funkcji przekazujemy dwie wartości, które w funkcji są przechowywane pod zmiennymi `x` i `y`, obie typu całkowitego `int`. Wartość tego samego typu będzie zwracana przez naszą funkcję. Polecenie `return` służy właśnie do zwracania wartości.

Może pojawić się sytuacja, że dla funkcji wymagającej argumentów nie zostaną one podane. Wówczas program zostanie przerwany. Aby temu zaradzić, możemy ustalić domyślne wartości dla zmiennych w funkcji. Oto przykład:

```
proc sum(x: int = 0, y: int = 0): int =  
  return x + y  
  
var result = sum()  
  
echo result
```

```
0  
-
```

Wywołując powyższą funkcję, nie podano żadnych argumentów. W efekcie, do obliczenia sumy zostały wykorzystane wartości domyślne dla `x` i `y` wynoszące 0.

## 6 Moduły

Moduły są to programy, które zawierają definicje funkcji. Funkcje takie usprawniają programowanie a dzięki temu, że znajdują się w swoich modułach (oddzielnych plikach), są odseparowane od właściwego kodu programu.

### 6.1 Instalacja i import modułów

Niektóre moduły są instalowane wraz z Nimem. Jeśli chcemy z nich skorzystać, wystarczy je zaimportować. Są jednak moduły dodatkowe, z których, kiedy chcemy skorzystać, trzeba je najpierw zainstalować.

Do instalacji dodatkowych modułów służy menedżer pakietów o nazwie `nimble`.

Zanim jednak będziemy chcieli wykonać jakieś operacje, warto sobie najpierw zsynchronizować listę modułów znajdującą się na naszym komputerze z tą, na serwerze:

```
nimble refresh
```

Wiec, gdy chcemy zainstalować nowy moduł:

```
nimble install nazwa_modułu
```

Dla przykładu, żeby zainstalować moduł o nazwie `Pixie` obsługujący grafikę, w terminalu wydajemy polecenie:

```
nimble install pixie
```

Gdy jednak chcemy odinstalować moduł:

```
nimble uninstall nazwa_modułu
```

Możemy także wyświetlić listę dostępnych do instalacji modułów.

```
nimble list
```

Jeśli jednak wiemy jakiego modułu szukamy to, zamiast pobierać informacje o wszystkich, możemy wyszukać konkretnego:

```
nimble search nazwa_modułu
```

W ramach krótkiego ćwiczenia napiszmy krótki program, który obliczy sinus kąta, wyrażonego w radianach. Do tego celu wykorzystamy funkcję `sin()` z modułu `math`, który wcześniej zaimportujemy. Jeśli moduł, który chcemy zaimportować pochodzi ze standardowej instalacji Nima, to dobrym zwyczajem jest (choć nie jest to niezbędne) dopisanie przed nazwą modułu `std`:

```
import std/math

echo PI
echo sin(PI)
```

```
3.141592653589793
1.224646799147353e-16
-
```

Moduł `math` oprócz definicji funkcji `sin()` posiada również zdefiniowaną wartość liczby `PI`, która została wyświetlona w powyższym przykładzie.

Oczywiście `sin( $\pi$ )` wynosi 0, jednak dokładność liczby  $\pi$  użyta w tym przykładzie, nie pozwala osiągnąć takiej wartości przy takiej precyzji wyświetlanego wyniku. Nie musimy się teraz tym przejmować.

Jeśli chcemy wykorzystać tylko jedną funkcję z danego modułu, wówczas możemy zaimportować tylko tę funkcję:

```
from std/math import sin
```

Jeśli znajdzie taka potrzeba można importować moduły z wykluczeniem konkretnych funkcji. Dla przykładu zaimportujemy moduł `math` ale bez funkcji `PI`:

```
import std/math except PI
```

Po zaimportowaniu modułu mamy bezpośredni dostęp do zdefiniowanych w nim funkcji. Możemy się do nich również odwoływać, poprzedzając ich nazwę nazwą modułu oddzieloną kropką o nazwy funkcji. Oto przykład:

```
import std/math

echo math.sqrt(9.0)
```

Można również stosować tzw. aliasy (`as`), dzięki którym będziemy mogli przechowywać zaimportowany moduł pod własną nazwą.

```
import std/math as m_functions

echo m_functions.sqrt(9.0)
```

Korzystanie z odpowiednich modułów przyspiesza pracę nad kodem i sprawia, że programuje się wygodniej. W dalszej części podręcznika skupimy się bardziej na kilku modułach, do których częściej będziemy sięgać.

## 6.2 Własne moduły

Istnieje możliwość tworzenia własnych modułów. Wymaga to jednak wcześniejszego poznania wiadomości dotyczących funkcji.

Nasz moduł, żeby można było z niego skorzystać, podczas kompilacji powinien znajdować się w tym samym folderze, co nasz główny program.

Prześledźmy poniższy przykład:

Tworzymy plik o nazwie `hi_module.nim`. W tym pliku tworzymy funkcję o nazwie `hi`, która będzie służyła do wyświetlania napisu `Hello world`. Oto zawartość pliku `hi_module.nim`:

```
proc hi*() =  
  echo "Hello world."
```

Gwiazdka (\*) umieszczona po nazwie funkcji a przed nawiasem pozwala na eksport tej funkcji. Gdybyśmy jej nie użyli, nie moglibyśmy tej funkcji zaimportować w głównym programie.

Zawartość głównego programu i wynik jego działania:

```
import hi_module  
  
hi()
```

```
Hello world.  
-
```

W kolejnym przykładzie stworzymy moduł, zawierający funkcję `sum`, sumującą dwie liczby całkowite przekazane do funkcji, jako argumenty. Tworzymy zatem plik główny `main.nim` oraz plik modułu o nazwie `calculate.nim` a wewnątrz tego pliku tworzymy funkcję `sum`:

```
proc sum*(a: int, b: int): int =  
  return a + b
```

Zawartość głównego pliku programu `main.nim`:

```
import calculate  
  
echo sum(3, 4)
```

7  
-

## 7 Sekwencje i moduł sequtils

### 7.1 Sekwencje

Sekwencje pozwalają przechowywać wiele wartości, przy czym nie trzeba z góry znać liczby tych wartości. Każda **sekwencja może przechowywać wartości tylko jednego typu**.

Stwórzmy sekwencję i przypiszmy do niej wartości:

```
var x = @[1, 2, 3]

echo x
echo type x
```

```
@[1, 2, 3]
seq[int]
-
```

Możemy także zadeklarować sekwencję bez dodawania do niej żadnych wartości:

```
var x = newSeq[string]()

echo x
echo type x
```

```
@[]
seq[string]
-
```

Zadeklarujmy nową sekwencję i dodajmy do niej jakieś wartości:

```
var x = newSeq[string]()
x.add("hello")
x.add("world")

echo x
```

```
@["hello", "world"]
-
```

Możemy też usunąć element sekwencji, gdy znamy jego indeks:

```
var x = @["hello", "world"]

echo x

x.del(1)

echo x
```

```
@["hello", "world"]
@["hello"]
-
```

W przypadku sekwencji możemy użyć takich samych konstrukcji, jakie zostały przedstawione dla tekstów w pierwszym przykładzie na stronie 14. Wówczas będziemy się odwoływać już nie do pojedynczych znaków, ale do całych elementów, z których składa się sekwencja.

## 7.2 Moduł sequtils

W module o nazwie `sequtils` znajduje się więcej funkcji służących do obsługi sekwencji.

Po zaimportowaniu modułu możemy połączyć kilka sekwencji w jedną, wykorzystując funkcję `concat()`, gdzie jako argumenty tej funkcji podajemy kolejne sekwencje:

```
import sequtils

var
  x = @[1, 2]
  y = @[3, 4]
  z = @[5, 6]
  c = concat(x, y, z)

echo c
```

```
@[1, 2, 3, 4, 5, 6]
-
```

Do dyspozycji mamy również funkcję o nazwie `count()` służącą do zliczania liczby wystąpień danego elementu sekwencji. Zliczmy, ile jest jedynek

w poniższej sekwencji:

```
import sequtils

var x = @[0, 1, 1, 2, -1]

echo x.count(1)
```

```
2
-
```

Sprawdźmy, na jakiej pozycji stoją minimalne i maksymalne wartości sekwencji:

```
import sequtils

var x = @[5, 3, -1, 2, 5]

echo x.minIndex
echo x.maxIndex
```

```
2
0
-
```

Jeśli maksymalnych lub minimalnych wartości jest więcej niż jedna, to funkcje `minIndex()` i `maxIndex()` zwracają indeksy pierwszych napotkanych wartości.

Wyświetlmy teraz wartości odpowiadające tym indeksom:

```
import sequtils

var x = @[5, 3, -1, 2, 5]

echo x[x.minIndex]
echo x[x.maxIndex]
```

```
-1
5
-
```



Funkcja `deduplicate()` służy do usunięcia duplikatów wartości sekwencji:

```
import sequtils

var x = @[1, 2, 2, 3, 3, 3]

echo x.deduplicate
```

```
@[1, 2, 3]
```

```
-
```

Z sekwencji możemy pobierać wartości, które spełniają określony przez nas warunek. Do tego służy funkcja `filter()`. Zatem użyjemy filtra do wyświetlenia tylko tych wartości sekwencji, które są mniejsze niż 3:

```
import sequtils

var x = @[-4, 7, 0, 3]

echo x.filter(proc(i: int): bool = i < 3)
```

```
@[-4, 0]
```

```
-
```

W powyższym przykładzie została użyta jednolinijkowa funkcja `proc()` sprawdzająca warunek i zwracająca wartość logiczną (`true` lub `false`), dzięki czemu funkcja `filter()` mogła zwrócić tylko te wartości, które spełniały ten wewnętrzny warunek.

Funkcja `keepIf()` daje taki sam rezultat co `filter()` z tą różnicą, że zmiany są zapisywane w aktualnej sekwencji:

```
import sequtils

var x = @[-4, 7, 0, 3]

x.keepIf(proc(i: int): bool = i < 3)

echo x
```

```
@[-4, 0]
```

```
-
```

Istnieje także możliwość podziału jednej sekwencji na kilka. Wówczas wewnątrz sekwencji są tworzone mniejsze, zawierające dane z pierwotnej sekwencji:

```
import sequtils  
  
var x = @[1, 2, 3, 4, 5, 6, 7, 8]  
  
echo x.distribute(2)
```

```
@[@[1, 2, 3, 4], @[5, 6, 7, 8]]
```

```
-
```

W celu zwielenokrotnienia sekwencji użyjemy funkcji `cycle()`, dla której argumentem jest liczba powtórzeń. Dla przykładu skopiujemy trzykrotnie wartości sekwencji:

```
import sequtils  
  
var x = @[1, 2, 3]  
  
echo x.cycle(3)
```

```
@[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
-
```

Dzięki funkcji `insert()` wstawimy wartości w miejsce o podanym indeksie. Funkcja przyjmuje dwa argumenty, gdzie jako pierwszy podajemy wartość, którą chcemy wstawić, a jako drugi numer indeksu, pod którym ta wartość ma zostać wstawiona. Dla przykładu wstawmy wartość 4 na pozycję o indeksie 2:

```
import sequtils  
  
var x = @[1, 2, 3]  
x.insert(4, 2)  
  
echo x
```

```
@[1, 2, 4, 3]
```

```
-
```

Gdybyśmy chcieli zmienić zawartość sekwencji, np. do każdej wartości dodać jakąś wartość, wówczas możemy skorzystać z funkcji `map()`. Dla przykładu pomnożmy każdą wartość w sekwencji przez dwa:

```
import sequtils  
  
var x = @[0, 1, 2, 3]  
echo x.map(proc(x: int): int = 2*x)  
  
echo x
```

```
@[0, 2, 4, 6]
```

```
-
```

## 8 Pętle

Pętle są konstrukcjami, które pozwalają na wielokrotne powtarzanie szeregu instrukcji w nich zawartych. W Nim mamy do dyspozycji dwa rodzaje pętli — `while` i `for`. Obie pętle mimo tego, że służą do tego samego — powtarzają część kodu — działają na różne sposoby.

### 8.1 Pętla `while`

Jest to pętla, która sterowana jest warunkiem. Wykonywać się będzie dopóty, dopóki warunek ten jest spełniany. Nie ma potrzeby określania na początku, ile razy pętla musi się wykonać. Dlatego pętla ta znajduje zastosowanie głównie w sytuacjach, kiedy nie jesteśmy w stanie na początku określić liczby powtórzeń.

Dla pętli `while` możemy przekazać informację, że warunek jest spełniony. Wówczas pętla staje się pętlą nieskończoną:

```
while true:
    echo "Hello world!"
```

Wynikiem działania powyższego kodu będzie nieskończone wyświetlanie tekstu `Hello world!`.

Pętla `while` z licznikiem:

```
var x = 1

while x <= 5:
    echo x
    x = x + 1
```

```
1
2
3
4
5
-
```

Zwiększanie lub zmniejszanie wartości zmiennej możemy realizować na kilka sposobów.

Różne sposoby inkrementacji i dekrementacji:

```

var x = 1

# Inkrementacja
x = x + 1
x += 1
inc x # inkrementuje tylko o 1

# Dekrementacja
x = x - 1
x -= 1
dec x # dekrementuje tylko o 1

```

Instrukcja `continue` wznowia działanie pętli, jednocześnie zaprzestając wykonywania dalszych instrukcji dla danego powtórzenia:

```

var x = 0

while x < 5:
    inc x
    if x == 3:
        continue
    echo x

```

```

1
2
4
5
-

```

Instrukcja `break` przerywa działanie pętli, w której została wywołana:

```

var x = 1

while true:
    echo x
    if x == 3:
        break
    inc x

```

```

1
2
3
-

```

## 8.2 Pętla for

Pętla `for` sprawdza się w sytuacjach, kiedy z góry wiemy, ile razy ta pętla ma się wykonać. Pętla `for` posiada wbudowany licznik, który możemy wykorzystać do odmierzania liczby powtórzeń.

Funkcja `countup()` może przyjąć trzy argumenty, kolejno: liczba początkowa, liczba końcowa, krok. Trzeci argument, czyli krok, jest opcjonalny. Tzn. nie musimy go podawać, gdy wartość ma się zmieniać o jeden — taka właśnie wartość jest ustawiona domyślnie.

```
for i in countup(1, 3, 1):  
    echo i
```

```
1  
2  
3  
-
```

Zmienna `i` w powyższym przykładzie jest to tzw. iterator pętli (łac. iteratio, itero — powtarzanie) — jest to zmienna sterująca działaniem pętli. Przyjmuje ona wartości zawarte w przedziale, określonym przez liczbę początkową i liczbą końcową. Obie te liczby należą do tego przedziału.

Analogicznie działa funkcja `countdown()` odliczająca od największej wartości do najmniejszej:

```
for i in countdown(3, 1, 1):  
    echo i
```

```
3  
2  
1  
-
```

Pętla ta dobrze sprawdza się podczas wyświetlania zawartości sekwencji. Tym razem będziemy iterować po jej poszczególnych elementach:

```
var colors = @["zielony", "czerwony", "niebieski", "biały"]

for color in colors:
    echo color
```

```
zielony
czerwony
niebieski
biały
-
```

Można także wyświetlić zawartość sekwencji wraz z numerami indeksów, pod którymi znajdują się jej elementy:

```
var colors = @["zielony", "czerwony", "niebieski", "biały"]

for index, color in colors:
    echo index, ". ", color
```

```
0. zielony
1. czerwony
2. niebieski
3. biały
-
```

W powyższym przykładzie na pierwszym miejscu stoi zmienna `index`, która pobiera kolejne numery indeksu. Nazwa tej zmiennej jest dowolna. Na drugim miejscu stoi zmienna `color`, która przyjmuje kolejne elementy sekwencji. Nazwa tej zmiennej także jest dowolna. Na końcu, po `in` znajduje się nazwa wyżej zadeklarowanej sekwencji.

## 9 Tablice

Tablice pozwalają przechowywać wiele wartości. Podobnie, jak sekwencje, muszą zawierać elementy jednego typu. To, co odróżnia tablice od sekwencji, to szybkość. Operacje wykonywane na tablicach z reguły są szybsze niż te wykonywane na sekwencjach. To jednak ma też swoje konsekwencje — wielkość tablicy musi zostać określona przed kompilacją programu. Czego nie musieliśmy robić w przypadku sekwencji.

W pierwszym przykładzie stwórzmy tablicę znakową, zawierającą trzy litery:

```
var arr = ['a', 'b', 'c']

echo arr.type
echo arr
```

```
array[0..2, char]
['a', 'b', 'c']
-
```

Dzięki funkcji `type()` uzyskaliśmy informacje o indeksach, pod którymi są przechowywane wartości oraz o typie danych przechowywanych przez tablicę.

Do poszczególnych elementów tablicy odwołujemy się tak samo, jak w przypadku sekwencji, podając numer indeksu, pod którym znajduje się dany element. Warto przypomnieć, że numery indeksów rozpoczynają się od 0.

```
var arr = ['a', 'b', 'c']

echo arr[0]
echo arr[^1]
```

```
a
c
-
```

Do wyświetlania zawartości tablicy często wygodnie jest wykorzystać pętlę `for`:



```
var arr = ['a', 'b', 'c']

for char in arr:
    echo char
```

```
a
b
c
-
```

Jeśli w chwili kompilacji programu nie mamy jeszcze danych, które możemy wstawić do tablicy, wówczas możemy ją zadeklarować a uzupełnić podczas realizacji programu.

```
var arr: array[3, int]

echo arr.type
echo arr
```

```
array[0..2, int]
[0, 0, 0]
-
```

Podczas deklaracji tablicy, tak jak już wcześniej zostało powiedziane, musimy podać, jak duża ma być tablica oraz jakiego typu dane będą w niej przechowywane. Jak możemy zaobserwować w powyższym przykładzie, deklaracja tablicy bez dodawania do niej wartości, powoduje uzupełnienie tablicy zadeklarowaną wcześniej liczbą następujących wartości:

- 0, w przypadku tablicy przechowującej liczby całkowite,
- 0.0, w przypadku tablicy przechowującej liczby zmiennoprzecinkowe,
- "", w przypadku tablicy przechowującej ciągi znaków,
- '\x00', w przypadku tablicy przechowującej pojedyncze znaki
- false, w przypadku tablicy przechowującej wartości logiczne.

Można także rozszerzyć deklarację tablicy o dodanie do niej wartości:

```

var arr1: array[3, int] = [1, 2, 3]
var arr2 = [1, 2, 3]

echo arr1.type
echo arr1
echo arr2.type
echo arr2

```

```

array[0..2, int]
[1, 2, 3]
array[0..2, int]
[1, 2, 3]
-

```

Oba sposoby stworzenia tablic (`arr1` i `arr2`) dają ten sam efekt.

Dodajmy teraz jakieś wartości do zadeklarowanej tablicy:

```

var names: array[3, string]
names[0] = "Adam"
names[1] = "Ewa"
names[2] = "Michał"

echo names

```

```

["Adam", "Ewa", "Michał"]
-

```

W dotychczasowych przykładach wykorzystywaliśmy tylko tablice jednowymiarowe. Teraz stwórzmy tablicę zawierającą dwie tablice, które będą miały po dwie liczby całkowite. Poniższy przykład pokazuje, w jaki sposób odwoływać się do poszczególnych indeksów tych tablic:

```

var x: array[2, array[2, int]] = [
    [1, 2],
    [3, 4]
]

echo x
echo x[0][0]
echo x[0][1]
echo x[1][0]
echo x[1][1]

```

```

[[1, 2], [3, 4]]
1
2
3
4
-

```

Jeśli tablicę uzupełniamy danymi przed kompilacją, wówczas nie musimy podawać szczegółów dotyczących jej rozmiaru i typu przechowywanych danych — są one pobierane na podstawie wprowadzonych wartości. Zatem zapiszmy ten sam przykład, co powyżej, w krótszej formie:

```

var x = [
    [1, 2],
    [3, 4]
]

echo x
echo x[0][0]
echo x[0][1]
echo x[1][0]
echo x[1][1]

```

```

[[1, 2], [3, 4]]
1
2
3
4
-

```

## 10 Tuple

Tuple, zwane inaczej krotkami pozwalają na przechowywanie wielu danych, jednak w odróżnieniu od sekwencji i tablic, pozwalają przechowywać dane różnych typów.

Oto przykładowa tupla, zawierająca dwie wartości — tekst i liczbę całkowitą:

```
var tup: (string, int) = ("Adam", 15)

echo tup
echo tup.type
```

```
("Adam", 15)
(string, int)
-
```

Jak można zauważyć w powyższym przykładzie, tupla jest reprezentowana przez nawiasy okrągłe.

Gdy tworząc krotkę, uzupełniamy ją od razu wartościami, wówczas nie musimy podawać typów danych — typy będą pobrane z wprowadzonych danych:

```
var tup = ("Adam", 15)

echo tup
echo tup.type
```

```
("Adam", 15)
(string, int)
-
```

Możemy się odwoływać do poszczególnych elementów tupli, podobnie, jak w przypadku sekwencji i tablic, podając numer indeksu:

```
var tup = ("Adam", 15)

echo tup[0]
echo tup[1]
```

```
Adam
15
-
```

Możemy także określić, gdzie dokładnie mają być przechowywane dane. Wyświetlmy od razu poszczególne elementy, tak, jak to robiliśmy do tej pory i drugim sposobem — wywołując wartości po nazwach, pod którymi przechowywane są one przechowywane:

```
var tup: tuple[name: string, age: int] = ("Adam", 15)

echo tup
echo tup[0]
echo tup[1]
echo tup.name
echo tup.age
```

```
(name: "Adam", age: 15)
Adam
15
Adam
15
-
```

Zadeklarujmy teraz krotkę, ale nie uzupełniajmy jej od razu, wartości dodamy już po jej utworzeniu:

```
var planet: tuple[name: string, water: bool]

planet.name = "Ziemia"
planet.water = true

echo planet
```

```
(name: "Ziemia", water: true)
-
```

## 11 Praca z plikami tekstowymi

### 11.1 Odczyt danych z pliku

Stwórzmy plik tekstowy o nazwie `plik.txt` i umieścmy w nim trzy linijki tekstu. Oto zawartość pliku tekstowego:

```
To jest pierwszy wiersz.  
To jest drugi wiersz.  
Koniec.
```

Plik z tekstem zapisujemy w tym samym katalogu, w którym znajdować się będzie plik z naszym programem.

Teraz z pomocą krótkiego kodu odczytamy jego zawartość i wyświetlimy w terminalu:

```
let text = readFile("plik.txt")  
  
echo text
```

```
To jest pierwszy wiersz.  
To jest drugi wiersz.  
Koniec.  
-
```

Do odczytu całej zawartości pliku tekstowego posłużyliśmy się funkcją o nazwie `readFile()`, dla której argumentem jest nazwa pliku lub pełna ścieżka do pliku, jeśli ten znajduje się w innym katalogu niż program. Zawartość pliku tekstowego została przypisana do stałej o nazwie `text` i wyświetlona w terminalu.

Jeśli spróbujemy otworzyć plik, który nie istnieje, otrzymamy komunikat o błędzie, a program zostanie przerwany. Możemy temu zaradzić, sprawdzając przed próbą otwarcia pliku, czy plik w ogóle istnieje. Tym razem skorzystamy z innego sposobu dostępu do pliku. Najpierw stworzymy zmienną `f` i przypiszemy jej typ plikowy `FILE`. Następnie z wykorzystaniem instrukcji warunkowej `if` spróbujemy otworzyć plik do odczytu — jest to domyślny sposób otwarcia pliku dla funkcji `open()`. Jeśli uda się uzyskać dostęp do pliku, zostanie zrealizowany blok kodu po instrukcji `if`. W przeciwnym razie blok kodu po instrukcji `else`. Do odczytu całej zawartości pliku tekstowego wykorzystamy funkcję `readAll()`:

```

var f: FILE

if open(f, "plik.txt"):
    echo "Hura! Plik istnieje!"
    echo f.readAll
    close(f)
else:
    echo "Plik nie istnieje..."

```

```

Hura! Plik istnieje!
To jest pierwszy wiersz.
To jest drugi wiersz.
Koniec.
-

```

Możemy także pobierać poszczególne linijki tekstu z pliku zamiast od razu całą jego zawartość. Linijki tekstu są odczytywane, licząc od pierwszej, funkcją `readLine()`. Każde jej wywołanie powoduje odczytanie kolejnej linijki z pliku. Odczytajmy pierwsze dwie linijki tekstu:

```

var f: FILE

if open(f, "plik.txt"):
    echo "Hura! Plik istnieje!"
    echo f.readLine
    echo f.readLine
    close(f)
else:
    echo "Plik nie istnieje..."

```

```

Hura! Plik istnieje!
Pierwsza linijka.
Druga linijka.
-

```

Otwierając plik funkcją `open()`, musimy pamiętać, żeby taki plik, po zakończonych operacjach na nim, zamknąć funkcją `close()`. Dzięki temu uwalniamy dostęp do tego pliku.

Możemy wczytać całą zawartość pliku, a potem podzielić ten tekst na poszczególne linijki. Do tego celu wykorzystamy funkcję `splitLines()` z modułu `strutils`:

```
import strutils

let
  text = readFile("plik.txt")
  lines = text.splitLines
  echo lines
```

```
@["Pierwsza linijka.", "Druga linijka.", "Koniec."]
```

W wyniku takiego podziału otrzymaliśmy sekwencję, której zawartością są poszczególne linijki tekstu.

## 11.2 Zapis danych do pliku

Żeby zapisać dane do pliku tekstowego, możemy użyć funkcji `writeFile()`, podając argumenty: nazwa pliku (lub pełna ścieżka do niego) do zapisu i tekst, który chcemy zapisać. Funkcja ta tworzy nowy plik, jeśli nie istnieje lub nadpisuje istniejący. Po zapisaniu danych do pliku zamyka go.

```
writeFile("plik.txt", "Hello world!")
```

Wróćmy jeszcze do funkcji `open()`, która dostarcza więcej opcji dostępu do pliku. Otworzymy sobie plik do zapisu, analogicznie, jak to zrobiliśmy w przykładzie powyżej:

```
var f = open("plik.txt", fmWrite)
f.write("Hello world!")

f.close
```

Poniżej znajduje się tabela z trybami otwierania plików (file mode, fm):

Tryb otwarcia pliku	Opis
fmRead	Tylko do odczytu.
fmWrite	Tylko do zapisu.
fmReadWrite	Do odczytu i zapisu. Jeśli plik nie istnieje, zostanie stworzony. Istniejący plik zostanie nadpisany.
fmReadWriteExisting	Do odczytu i zapisu. Jeśli plik nie istnieje, nie zostanie stworzony. Istniejący plik nie zostanie nadpisany.
fmAppend	Tylko do zapisu. Dane zostaną dołączone na końcu pliku.



Zatem, żeby dopisać nowy tekst do zawartości istniejącego pliku wykorzystamy `fmAppend`:

```
var f = open("plik.txt", fmAppend)
f.write("\nAla ma kota")

f.close
```