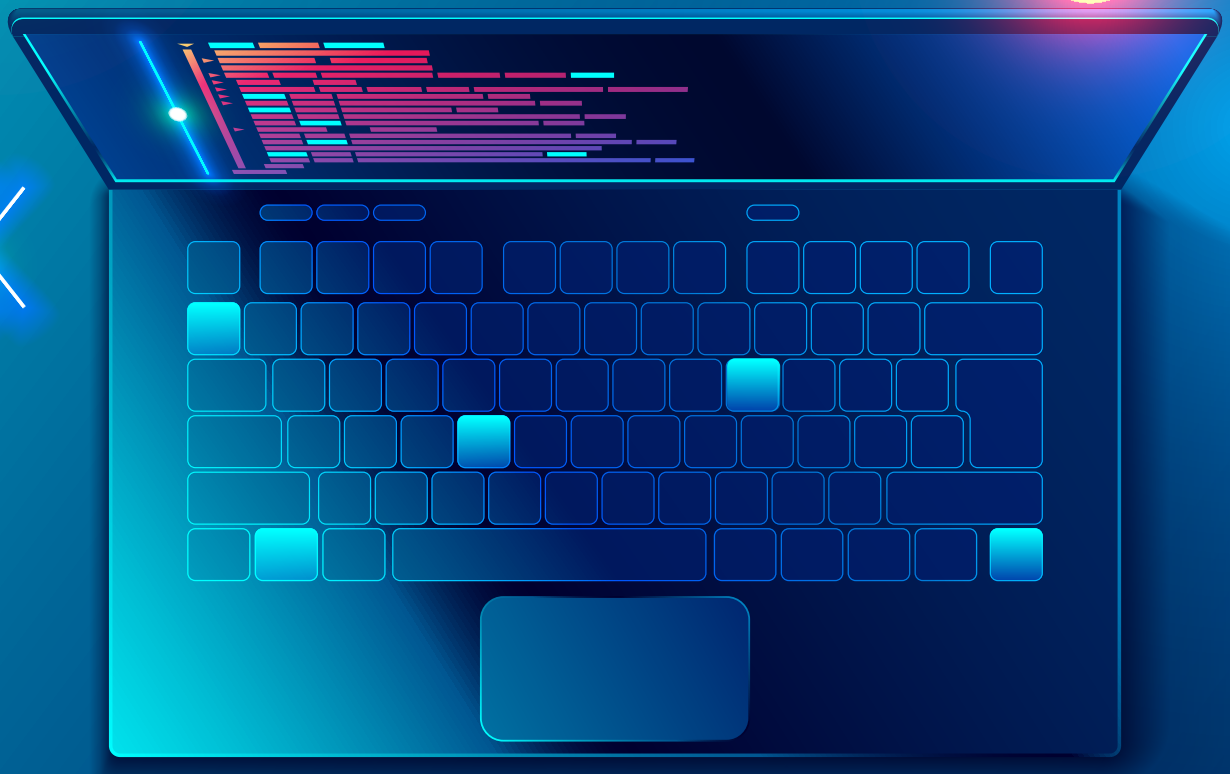
GO

PODSTAWY JĘZYKA



LINUX MAGAZINE

Podstawy języka Go

Aby móc programować w języku Go, trzeba najpierw pobrać odpowiedni pakiet z witryny projektu (https://go.dev/dl/). Jeśli pracujemy w Linuksie, możemy też zainstalować Go za pomocą menedżera pakietów (np. poleceniem *apt install golang* w Debianie).

Przykładowy program

Spójrzmy na poniższy przykład:

```
package main

import "fmt"

func main() {
      fmt.Println("Witaj, świecie!")
}
```

Przedstawia on prosty, lecz kompletny program w języku Go, który możemy uruchomić na dwa sposoby. Przede wszystkim należy go zapisać w pliku z rozszerzeniem *.go*, np. *hello.go*. Następnie możemy go albo uruchomić bezpośrednio:

```
go run hello.go
```

…lub najpierw skompilować:

```
go build hello.go
```

W wyniku uruchomienia powuższego polecenia w bieżącym katalogu pojawi się plik binarny (*hello.exe* w Windows; *hello* w Linuksie i macOS-ie), który można uruchomić na danym komputerze – lub innym o tej samej architekturze.
Program rozpoczyna się od słowa kluczowego *package* (dosłownie „pakiet”), które musi znaleźć się na początku każdego pliku z kodem źródłowym Go. Nazwa *main* oznacza, że wygenerowany zostanie program wykonywalny, a nie biblioteka.
Kolejne polecenie, *import*, oznacza, że importujemy funkcje znajdujące się w pakiecie *fmt*. Jest to niezbędne, ponieważ później użyjemy funkcji fmt.Println, która pochodzi z tego pakietu. Zwróćmy uwagę, że nazwa modułu znajduje się w cudzysłowie.
Dalej mamy słowo kluczowe *main*, które służy do definiowania funkcji. Tą funkcją jest *main()* – główna funkcja w programie. Definicja funkcji składa się ze słowa kluczowego *func*, nazwy funkcji (w tym przypadku *main*), opcjonalnie argumentów w nawiasach okrągłych oraz treści funkcji w nawiasach klamrowych. Sama treść funkcji składa się w tym wypadku z pojedynczego wiersza.

Wiersz ten zawiera inną funkcję, której nazwa składa się z dwóch części: *fmt* i *Println()*. Pierwsza część wskazuje na nazwę pakietu, który zaimportowaliśmy wcześniej, zaś druga to nazwa funkcji, która została zaimplementowana w tym pakiecie. Jest to funkcja *Println*, która służy do wyświetlania danych na ekranie, dodając na końcu znak nowego wiersza.

Zwróćmy uwagę, że na końcu wierszy nie stawiamy średników.

Zmienne

„Zmienna" to zasadniczo adres w pamięci komputera, któremu możemy przypisać nazwę oraz typ. Np. ten sam adres pamięci zawierający bity *01100100* można określić jako liczbę dziesiętną *100*, liczbę szesnastkową *64* albo literę *d*.
W poniższym przykładzie deklarujemy zmienną *liczba* i przypisujemy jej typ *int*, który oznacza liczby całkowite, używając słowa kluczowego *var*:

```
var liczba int
```

Następnie przypisujemy zmiennej *liczba* wartość 64:

```
liczba = 64
```

Możemy też użyć notacji skróconej, od razu przypisując wartość zmiennej, a na tej podstawie Go sam wywnioskuje, jaki typ powinien być przypisany. W tym przypadku nie używamy słowa kluczowego *var*, natomiast zamiast znaku równości używamy sekwencji *:=*.

```
liczba_zmiennoprzecinkowa := 8.0
```

Spójrzmy na kompletny program, w którym używamy obu konstrukcji, po czym wyświetlamy obie liczby na ekranie:

```
package main

import "fmt"

func main() {
      var liczba int
      liczba = 64
      liczba_zmiennoprzecinkowa := 8
      fmt.Println(liczba, liczba_zmiennoprzecinkowa)
}
```

Po uruchomieniu programu bezpośrednio (*go run t1.go*) lub po kompilacji (*go build t1.go*, *./t1* lub *t1.exe*) powinniśmy otrzymać poniższy rezultat:

```
64 8
```

Jak widzimy, funkcja *Println()* oddzieliła obie zmienne znakiem spacji, zaś na koniec dodała znak nowego wiersza. Czasem może nam zależeć na bardziej precyzyjnej kontroli wyświetlania – służy do tego funkcja *Printf()* z tego samego pakietu *fmt*.

O ile Println stosuje formatowanie domyślne dla każdej zmiennej, Printf umożliwia wymuszenie formatowania za pomocą odpowiednich sekwencji formatujących rozpoczynających się od znaku procenta. Na przykład domyślna reprezentacja danej zmiennej to *%v*, reprezentacja binarna (czyli w systemie dwójkowym) to *%b*, nazwa typu danej zmiennej to *%T*, zaś notacja naukowa dla zmiennych zmiennoprzecinokwych (czyli typu *float*) to *%e*.
Zatem przykładowy program:

```
package main

import "fmt"

func main() {
	var liczba int
	liczba = 64
	lz := 64.0
	fmt.Printf("%v %b %T %e\n", liczba, liczba, liczba, liczba)
	fmt.Printf("%v %b %T %e\n", lz, lz, lz, lz)
}
```

Wyświetli poniższe dane:

```
64 1000000 int %!e(int=64)
64 4503599627370496p-46 float64 6.400000e+01
```

Jak widzimy, zmienne typu całkowitego (*int*) nie obsługują poprawnie znacznika formatowania notacji naukowej *%e*.
Go obsługuje wiele typów danych, z których najważniejsze to:

- *bool* – może przyjmować jedynie wartości *true* (prawda) lub *false* (fałsz)
- *int* – liczby całkowite, np. *5* czy *-8*
- *string* – napisy, np. *To jest napis* czy *bęcki*
- *float32* – liczby zmiennoprzecinkowe, np. *2.0* czy *3.141*
- *complex64* – liczby zespolone, np. *1+2i*

Pętla for

Pętla for umożliwia wykonywanie znajdujących się wewnątrz niej poleceń określoną liczbę razy, do momentu, kiedy wystąpi zdefiniowany wcześniej warunek. W bardzo prostej wersji pętla for może zawierać jedynie ten warunek. Wynikiem działania poniższego programu:

```
package main

import "fmt"

func main() {
	i := 1
	for i <= 3 {
		fmt.Println(i)
		i += 1
	}
}
```

Będzie poniższa sekwencja:

```
1
2
3
```

Możemy też opuścić warunek – jeśli z powyższego kodu usuniemy *i <= 3*, pętla będzie działała w nieskończoność. Bardziej tradycyjna postać pętli wygląda jak niżej:

```
package main

import "fmt"

func main() {
	for i := 1; i <= 3; i++ {
		fmt.Println(i)
	}
}
```

Wynik działania programu będzie identyczny jak poprzednio – poszczególne wartości zmiennej *i*, od początkowej *1* do końcowej *3*.

Instrukcje warunkowe if/else

Konstrukcja if/else służy do kontroli przepływu programu. W najprostszym wariancie po słowie kluczowym *if* następuje warunek; jeśli warunek ten jest spełniony, wykonywane są instrukcje znajdujące się w nawiasach klamrowych bezpośrednio po warunku. Jeśli chcemy, możemy dodać opcjonalny blok *else*, który zostanie wykonany, jeśli warunek nie zostanie spełniony.
Poniższy program wyświetli napis *To prawda!*:

```
package main

import "fmt"

func main() {

	a := true
	if a == true {
		fmt.Println("To prawda!")
	} else {
		fmt.Println("To nieprawda!")
	}
}
```

Tablice i wycinki

Tablice (array) w Go służą do przechowywania sekwencji zmiennych tego samego typu. Np. aby zadeklarować jednowymiarową tablicę składającą się z pięciu liczb całkowitych, użyjemy konstrukcji

```
var a [5]int
```

Spójrzmy na poniższy przykład:

```
package main

import "fmt"

func main() {

	var a [3]int
	fmt.Println(a)
	var b [3][3]int
	fmt.Println(b)

}
```

Deklaruje on dwie tablice, a i b. Pierwsza jest jednowymiarowa, zaś druga – dwuwymiarowa. Obie przechowują wyłącznie wartości typu *int*. Wynik działanie programu to:

```
[0 0 0]
[[0 0 0] [0 0 0] [0 0 0]]
```

Tablice w Go mają jedną zasadniczą wadę: mają stałą wielkość. Jeśli zadeklarowaliśmy, że dana tablica ma 5 elementów, to nie możemy tej liczby ani zwiększyć, ani zmniejszyć. Dlatego w praktyce zamiast tablic częściej stosuje się tzw. wycinki (bądź plasterki – *slices*), które można modyfikować.
Do poszczególnych elementów tablic i plasterków odwołujemy się za pomocą nawiasów kwadratowych, w których umieszczany numer elementu, pamiętając, że pierwszy element ma numer 0, drugi to 1 itd. Domyślnie wszystkie elementy liczbowe mają wartość 0. Kolejne elementy dodajemy do wycinków za pomocą funkcji *append()*, natomiast za pomocą *len()* możemy sprawdzić długość tablicy bądź wycinka.
Poniższy program…

```
package main

import "fmt"

func main() {
	a := make([]int, 3)
	for i := 0; i < len(a); i++ {
		fmt.Println(a[i])

	}
```

```
	for i := 0; i < len(a); i++ {
		a[i] = i

	}

	a = append(a, 10)

	fmt.Println(a)
}
```

…wygeneruje następujące dane wyjściowe:

```
0
0
0
[0 1 2 10]
```

Zawiera on dwie pętle *for*: w pierwszej wyświetlamy poszczególne elementy wycinka, zaś w drugiej – przypisujemy im kolejne wartości od 0 do 3.
Co więcej, za pomocą nawiasów klamrowych możemy odwoływać się nie tylko do pojedynczych elementów, ale również całych zakresów – wykorzystywana jest konstrukcja *wycinek[początek:koniec]*, przy czym element *początek* jest zawsze włączany, natomiast element *koniec* – nigdy. Np. pierwszy i drugi element wycinka a to *[0:2]*, a nie *[0:1]*.
W wyniku uruchomienia poniższego programu:

```
package main

import "fmt"

func main() {
	a := make([]int, 5)

	for i := 0; i < len(a); i++ {
		a[i] = i

	}

	fmt.Println(a, a[1:3], a[:3], a[3:])
}
```

Uzyskamy wynik:

```
[0 1 2 3 4] [1 2] [0 1 2] [3 4]
```

Mapy

Mapy, czyli tablice asocjacyjne, to bardzo przydatny typ danych, zwany również słownikiem lub tablicą skojarzeniową. Jest to typ danych składający się z kluczy i przypisanych im wartości. Mapę tworzymy za pomocą *make(map[typ-klucza]typ-wartości)*.
Przykład utworzenia mapy znajdziemy poniżej:

```
package main

import "fmt"

func main() {
	a := make(map[string]int)
	a["Adam"] = 650123123
	a["Robert"] = 651123123

	fmt.Println(a)
	fmt.Println(a["Adam"])
}
```

W wyniku uruchomienia programu uzyskamy poniższe wyjście:

```
map[Adam:650123123 Robert:651123123]
650123123
```

Aby wyświetlić wszystkie klucze i wartości, najprościej jest użyć konstrukcji *range*:

```
package main

import "fmt"

func main() {
	a := make(map[string]int)
	a["Adam"] = 650123123
	a["Robert"] = 651123123

	for k, w := range a {
		fmt.Println(k, w)
	}
}
```

Powyższy program wyświetli następujące dane:

```
Adam 650123123
Robert 651123123
```

Funkcje

Funkcja to logiczna część kodu, z reguły wykonująca jedno określone zadanie. Funkcje często przyjmują argumenty i zwracają wartości.
W Go definicja funkcji zaczyna się do słowa kluczowego *func*, po którym umieszczamy parametry w nawiasach okrągłych, przy czym każdy parametr musi mieć określony typ. Uwaga: po liście argumentów następuje typ wartości zwracanej przez funkcję, po czym sama funkcja. Wartość funkcji zwracana jest za pomocą słowa kluczowego *return*.

Poniższy program wyświetli liczbę *4*:

```
package main

import "fmt"

func dodaj(a int, b int) int {
	return a + b
}

func main() {
	fmt.Println(dodaj(2, 2))
}
```

Co ciekawe, w Go funkcja może zwracać więcej niż jedną wartość, np. poniższy program zwróci wynik *4 0*:

```
package main

import "fmt"

func suma_roznica(a int, b int) (int, int) {
	return a + b, a - b
}

func main() {
	fmt.Println(suma_roznica(2, 2))
}
```

Go obsługuje też zmienną liczbę argumentów za pomocą elipsy (…). Poniższy program wyświetli liczbę *6*:

```
package main

import "fmt"

func suma(liczby ... int) int {
	s := 0
	for _, n := range liczby {
		s += n
	}
	return s
}
```

```
func main() {
	fmt.Println(suma(1, 2, 3))
}
```

Gorutyny

Go ułatwia tworzenie lekkich wątków za pomocą słowa kluczowego *go*. Poniższy program przedstawia uruchomienie przedstawionej wcześniej funkcji sumujących bezpośrednio i za pomocą *go()*. Ponieważ gorutyna działa współbieżnie, musimy na nią zaczekać – w przykładzie poniżej używamy do tego metody *Sleep* z pakietu *time*:

```
package main

import (
	"fmt"
	"time"
)

func suma(liczby ...int) {
	s := 0
	for _, n := range liczby {
		s += n
	}
	fmt.Println(s)
}

func main() {
	suma(1, 2, 3)
	go suma(2, 3, 4)
	suma(3, 4, 5)
	time.Sleep(time.Second)
}
```

LINUX MAGAZINE

2LM0003