POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

Podstawy GO

Przydatne linki

Wprowadzenie i dokumentacja

- Oficjalna dokumentacja i tutoriale
- A Tour of Go
- Go by Example

Go w przeglądarce

- Go Playground

Nauka przez ćwiczenia

- Learn Go with Tests
- Exercism - Go Track

Variables & Strings

```go
import "fmt"

func main() {
    var hello string = "Hello"
    name := "PJATK"

    fmt.Println(hello, name, "!")
    // => Hello PJATK !
}
```

- Dwa sposoby deklaracji zmiennych: `var` lub `:=`
- Inferencja typów - koniec z `Button button = new Button(„Button");`.

Variables & Strings

```go
var name = "World"

func main() {
    name := "PJATK"
    fmt.Println("Hello", name, "!")
    // => Hello PJATK !

    printGlobal()
    // => Hello World !
}

func printGlobal() {
    fmt.Println("Hello", name, "!")
}
```

- Zmienne globalne - tylko `var`
- Zmienna lokalna „przysłania” globalną

Variables & Strings

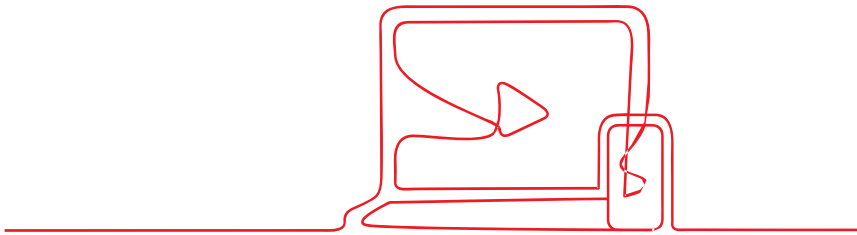
```go
var emptyVariable string
var unused string

func main() {
    fmt.Println(emptyVariable)
    // =>

    var compilationError string
    // compilationError declared but not used
}
```

- Nazwy zmiennych zwyczajowo zapisuje się camelCasem
- Deklaracja zmiennej bez podania wartości inicjalizuje ją z domyślną wartością dla danego typu.
- Domyślna wartość typu string to pusty ciąg znaków.
- Nieużycie zadeklarowanej zmiennej lokalnej to błąd uniemożliwiający kompilację.

Variables & Strings

```
greeting := "hello" + "pjatk"
fmt.Println(greeting)
// => hellopjatk

greeting = fmt.Sprintf("%s %d!", "Hello", "2023")
fmt.Println(greeting)
// => Hello 2023!
```

- Operator dodawania (+) łączy ze sobą ciągi znaków.
- Interpolacja stringów pozwala na dowolne ich formatowanie.
- Dostępne symbole można znaleźć w dokumentacji.



Variables & Strings

```
fmt.Println("\xbd\xb2")
// => ??

fmt.Println("Hello, 世界 😉")
// => "Hello, 世界 😉"

世界 := "OK"
fmt.Println(世界)
// => OK

emoji := "😱"
fmt.Println(len(emoji))
// => 4
```

- Stringi to ciągi dowolnych bajtów.
- Większość funkcji operujących na stringach zakłada UTF-8.
- Kod źródłowy to zawsze UTF-8.
- W nazwach funkcji oraz zmiennych dozwolony jest ograniczony zakres znaków.
- Funkcja len zwraca liczbę bajtów, nie liczbę znaków.

Integers

```
var zero int
fmt.Printf("%d\n", zero)
// => 0

fmt.Println(zero + -1)
// => -1

fmt.Println(5 / 3)
// => 1

fmt.Println(7 % 5)
// => 2
```

- Domyślna wartość dla liczby całkowitej to 0.
- Dostępne operacje to: dodawanie (+), odejmowanie (-), mnożenie (*), dzielenie (/) oraz modulo (%).

Integers

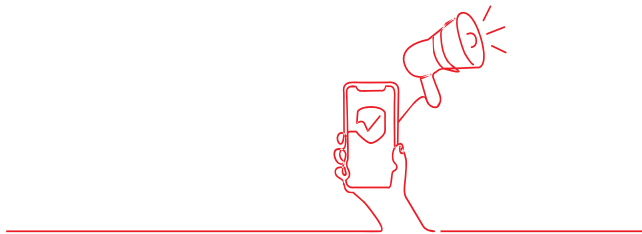
```go
var signedInt int8
fmt.Println(signedInt - 1)
// => -1

signedInt = -128
fmt.Println(signedInt - 1)
// => 127

var unsignedInt uint8 = 255
fmt.Println(unsignedInt + 1)
// => 0
```

- Dostępne typy to: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`.
- Rozmiar `int` oraz `uint` zależy od architektury i wynosi 32 albo 64 bity.
- Są jeszcze aliasy: `uintptr` (`uint`), `byte` (`uint8`) oraz `rune` (`int32`).
- Należy pamiętać o integer overflow i underflow.

Floats

```go
var zero float64
fmt.Printf("%f\n", zero)
// => 0.000000

pi := 3.14159265359
fmt.Printf("%T -> %.2f\n", pi, pi)
// => float64 -> 3.14
```

- Domyślna wartość to 0.
- Dwa typy: `float32` oraz `float64`.
- Domyślny typ (gdy go nie sprecyzujemy) to `float64`.



Floats

```
float := 3.14
integer := 10

fmt.Println(float * float64(integer))
// => 31.400000000000002

var result int
result = int(math.Pow(3, 10))
fmt.Println(result)
// => 59049
```

- Precyzja jest ograniczona - należy uważać na „nierówne” wyniki.
- Konwersji typów trzeba dokonać jawnie, operacje na różniących się typach nie są dozwolone.

Floats

```
math.Abs(-1)
// => 1

negative := -1
math.Abs(negative)
// => cannot use negative (variable of type int)
//    as type float64 in argument to math.Abs
```

- Literały (wartości zapisane bezpośrednio w kodzie) nie mają określonego typu i przyjmują go w zależności od potrzeb (jeśli konwersja jest możliwa).

Constants

```
const (
    lightSpeed = 299792458
    pi = 3.14159265359
)
```

- Stałe wartości deklarujemy przy użyciu `const`.
- Zarówno stałe jak i zmienne można deklarować “hurtowo”, używając nawiasów.

Booleans

```
var truth bool
fmt.Println(truth)
// => false

fmt.Println(true && false)
// => false

fmt.Println(true || false)
// => true

fmt.Println(!false)
// => true
```

- Domyślna wartość to `false`.
- Dostępne operatory: && (and), ll (or), ! (not).
- Nie występują „truthy/falsy values”.

Functions

```go
func main() {
    fmt.Println(sumToString(2.345433333, 3.33321, 5))
    // -> 5.67864

    function := sumToString
    fmt.Println(function(6.12345, 1.654321, 3))
    // -> 7.77777
}

func sumToString(n1, n2 float64, a int) string{
    sum := n1 + n2
    return strconv.FormatFloat(sum, 'f', 5, 64)
}
```

- Funkcje definiujemy za pomocą *func*.
- Funkcję możemy przypisać do zmiennej, przekazać do innej funkcji

If/Else

```
if true || (true && false) {
    fmt.Println("IF")
} // => IF

if 2+2 != 4 {
    fmt.Println("IF")
} else {
    fmt.Println("ELSE")
}
// => ELSE
```

- Nawiasy wokół warunku są opcjonalne.
- Wykonywany jest pierwszy blok od góry, którego warunek zostanie spełniony.

If/Else

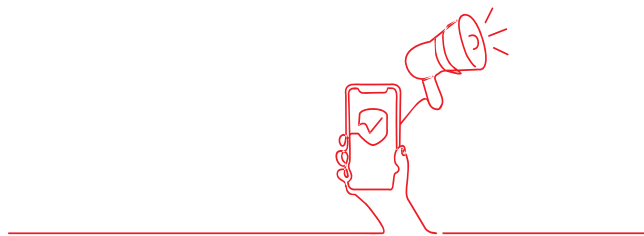
```go
if pi := math.Pi; pi < 3 {
    fmt.Println("?")
} else if pi >= 3 {
    fmt.Printf("%.2f\n", pi)
}
// => 3.14
```

- Bloki `else if` oraz `else` nie są wymagane.
- Zmienna zadeklarowana w bloku `if` jest dostępna w tym bloku oraz następujących po nim blokach `else if`/`else`.

If/Else

```
if n := rand.Intn(10000); n == 0 {
    fmt.Println("zero")
} else if n < 10 {
    fmt.Println("less than 10")
} else if n < 100 {
    fmt.Println("less than 100")
} else if n > 9000 {
    fmt.Println("IT'S OVER 9000!")
} else {
    fmt.Println("whatever")
}
```

- Blok *else if* można powtarzać wielokrotnie.



Switch

```
n := rand.Intn(10000)
switch {
case n < 10:
    fmt.Println("less than 10")
case n < 100:
    fmt.Println("less than 100")
case n > 9000:
    fmt.Println("IT'S OVER 9000!")
default:
    fmt.Println("whatever")
}
```

- W najprostszej wersji każdy case switcha to wyrażenie logiczne.
- Wykonywany jest pierwszy od góry „prawdziwy” blok.
- Ostatni blok to opcjonalny `default`, który wykona się wtedy, gdy nie zostanie wykonany żaden z wcześniejszych.

Switch

```
switch n := rand.Intn(10); n {
case 0:
    fmt.Println("zero")
case 1, 2:
    fmt.Println(„one or two”)
default:
    fmt.Println("whatever")
}
```

- Jeśli poleceniu `switch` przekażemy wartość, to wykonany zostanie pierwszy blok, który jest równy tej wartości.
- Przypisanie wartości jest opcjonalne.

For

```
for counter := 0; counter < 3; counter++ {
    fmt.Printf("%d..", counter)
}
// => 0..1..2..
```

- Inicjalizacja; warunek; inkrementacja.
- Inicjalizacja wykonywana jest przed pierwszym wykonaniem pętli.
- Warunek sprawdzany jest przed każdym wykonaniem pętli.
- Inkrementacja wykonywana jest po każdym przejściu pętli.

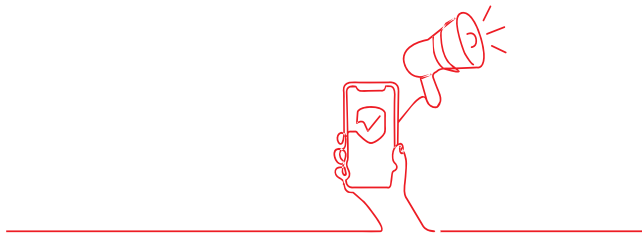
For

```
counter := 0
for counter < 3 {
    fmt.Printf("%d..", counter)
    counter++
}

for counter := 0; counter < 3; counter++ {
    fmt.Printf("%d..", counter)
}

for counter := 0; counter < 3; {
    fmt.Printf("%d..", counter)
    counter++
}

counter = 0
for ; counter < 3; counter++ {
    fmt.Printf("%d..", counter)
}
```

For

```
for true {
    fmt.Println("infinite loop")
}

for false {
    fmt.Println("this will never execute")
}

for {
    fmt.Println("infinite loop")
}
```

- Uwaga na nieskończone pętle.
- Pętla `for` bez warunku jest równoznaczna z pętlą `for true`.



For

```
counter := 1
for {
    if counter%5 == 0 {
        break
    } else {
        fmt.Printf("%d..", counter)
        counter++
        continue
    }
    fmt.Println("unreachable code")
}
// => 1..2..3..4..
```

- Polecenie `break` wychodzi z pętli.
- Polecenie `continue` kończy obecną iteracje i zaczyna następną.
- Widoczny kod to przykład źle przemyślanego kodu - nie piszcie tak.

For

```
for counter := 1; counter%5 != 0; counter++ {
    fmt.Printf("%d..", counter)
}
// => 1..2..3..4..
```

→ Ten kod robi dokładnie to samo.

For

```go
hello := "Hello, 世界 😉"

fmt.Println(len(hello))
// => 18

length := 0
for range hello {
    length++
}
fmt.Println(length)
// => 11
```

- Przy użyciu `range` możemy iterować po kolekcjach - stringach, slice'ach, mapach, intach.
- Iterowanie po stringach zwraca runy, które mniej więcej odpowiadają widocznym znakom.

For

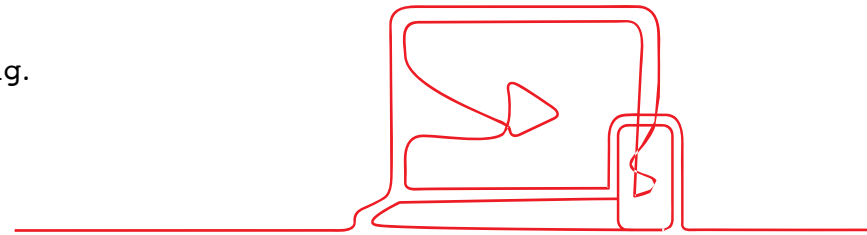
```
length := 0
for range 10 {
    length++
}
fmt.Println(length)
// => 10
```

→ Od wersji Go 1.22 istnieje możliwość iterowania po intach (range zwraca tylko index)

For

```go
for index, value := range "Hello 世界" {
    fmt.Printf("%d: %s\n", index, string(value))
}
// => 0: H
// => 1: e
// => 2: l
// => 3: l
// => 4: o
// => 5:
// => 6: 世
// => 7: 界
```

- `range` zwraca indeks oraz wartość.
- Typ `rune` to alias `int32`, stąd zamiana z powrotem na `string`.



For

```
for index := range "abc" {
    fmt.Println(index)
}
// => 0
// => 1
// => 2

start := 3
for range "abc" {
    fmt.Println(start)
    start--
}
// => 3
// => 2
// => 1
```

- Można pominąć wartość, jak i wartość i indeks.

For

```go
for _, value := range "hello" {
    fmt.Println(value)
}
// => 104
// => 101
// => 108
// => 108
// => 111
```

- Indeksu pominąć nie można - jeśli nie jest potrzebny, należy użyć znaku underscore (_).

Arrays & Slices

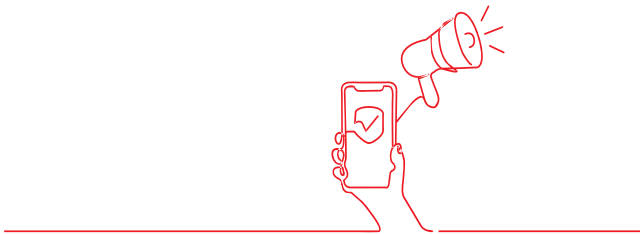
```go
var array [10]int
fmt.Println(array)
// => [0 0 0 0 0 0 0 0 0 0]

var slice []int
fmt.Println(slice)
/// => []
```

- Array - stała liczba elementów.
- Slice - nie ma stałego rozmiaru, powiększa się w miarę potrzeb.

Arrays & Slices

```go
var array [10]int = [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
fmt.Println(array)
// => [1 2 3 4 5 6 7 8 9 0]

var slice []int = array[3:5]
fmt.Println(slice)
// => [4, 5]
```

- Brakujące wartości uzupełniane są domyślną wartością.
- Slice oraz array możemy dowolnie „ciąć”.



Arrays & Slices

```
slice := []string{"a", "b", "c"}

slice[0] = "d"

fmt.Println(slice[0])
// => d
```

- Możemy przypisać i odczytać konkretną wartość używając indeksu w nawiasach kwadratowych.
- Nie ma negatywnych indeksów.

Arrays & Slices

```
slice1 := []int{1, 2}
slice2 := append(slice1, 3, 4)
fmt.Println(slice2)
// => [1 2 3 4]

fmt.Println(slice1)
// => [1 2]

fmt.Println(append(slice1, slice2...))
// => [1 2 1 2 3 4]
```

- `append` zwraca nowy slice z elementami dodanymi na końcu.
- Oryginalny slice pozostaje niezmieniony.
- Jeśli chcemy połączyć dwa istniejące slice'y, należy użyć ..., by zamienić slice na oddzielne elementy.

Arrays & Slices

```go
var slice []int

fmt.Println("Length:", len(slice))
// => Length: 0
fmt.Println("Capacity:", cap(slice))
// => Capacity: 0
```

- Slice ma zarówno długość, jak i pojemność.
- Długość oznacza aktualną liczbę elementów.
- Pojemność to liczba elementów, po której nastąpi powiększenie.

Arrays & Slices

```go
for slice := []int{}; len(slice) < 10; slice = append(slice, 1) {
    fmt.Println("len:", len(slice), "cap", cap(slice))
}
// => len: 0 cap: 0
// => len: 1 cap: 1
// => len: 2 cap: 2
// => len: 3 cap: 4
// => len: 4 cap: 4
// => len: 5 cap: 8
// => len: 6 cap: 8
// => len: 7 cap: 8
// => len: 8 cap: 8
// => len: 9 cap: 16
```

- Przy każdym powiększeniu pojemność rośnie dwukrotnie.
- https://go.dev/play/p/mKF51Gkl8a_R

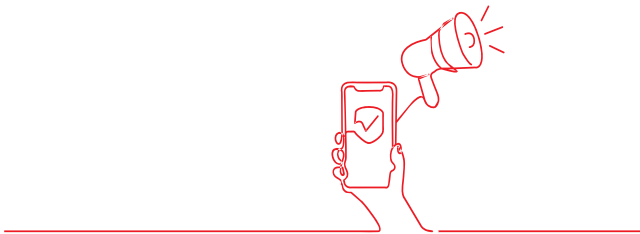
Arrays & Slices

```
const threshold = 256
if oldCap < threshold {
    return doublecap
}
// Transition from growing 2x for small slices
// to growing 1.25x for large slices. This formula
// gives a smooth-ish transition between the two.
newcap += (newcap + 3*threshold) >> 2
```

Arrays & Slices

```
slice := make([]int, 2, 10)
fmt.Println(slice)
// => [0 0]

fmt.Println("len:", len(slice), "cap:", cap(slice))
// => len: 2 cap: 10
```

- Funkcja make pozwala utworzyć slice o predefiniowanej długości i/lub pojemności.



Arrays & Slices

```go
slice1 := []int{1, 2, 3, 4, 5}
slice2 := slice1[1:4]
fmt.Println(slice2)
// => [2, 3, 4]

slice1[2] = -1
fmt.Println(slice2)
// => [2, -1, 4]
```

→ Uwaga na niespodziewane zmiany!

Arrays & Slices

```go
copyFrom := []int{1, 2, 3, 4, 5}
copyTo := make([]int, len(copyFrom))
copy(copyTo, copyFrom)
copyFrom[2] = -1
fmt.Println(copyTo)
// => [1 2 3 4 5]
```

- Skopiowanie slice'a zabiezpiecza nas przed zmianami.

Arrays & Slices

```go
copyFrom := [][]int{{1, 2, 3}, {3, 4, 5}}
copyTo := make([][]int, len(copyFrom))
copy(copyTo, copyFrom)

copyFrom[0] = []int{0, 0, 0}
fmt.Println(copyTo)
// => [[1 2 3] [3 4 5]]

copyFrom[1][0] = -1
fmt.Println(copyTo)
// => [[1 2 3] [-1 4 5]]
```

→ Zagnieżdżone struktury (slice, map, struct) wciąż mogą się zmienić.

Arrays & Slices

```go
var slice []int

fmt.Println(slice)
// => []
fmt.Println(len(slice))
// => 0

fmt.Println(slice == nil)
// => true
```

- Domyślną wartością slice'a jest nil.

Arrays & Slices

```go
var slice []int
buf, _ := json.Marshal(slice)
fmt.Println(string(buf))
// => null

slice = []int{}
buf, _ = json.Marshal(slice)
fmt.Println(string(buf))
// => []
```

- Można się na to naciąć przy serializacji do JSON-a.
- Najbezpieczniej jest zainicjalizować pusty slice poprzez `[]int{}` lub `make([]int, 0)`.

Arrays & Slices

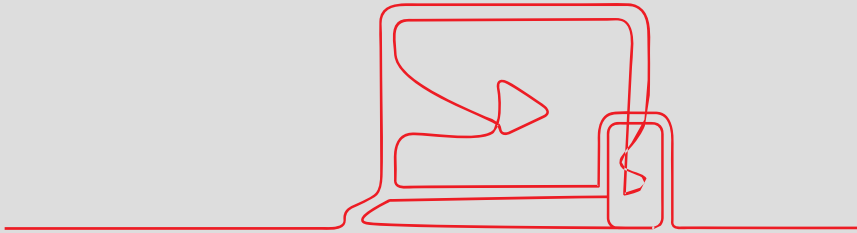
Usuwanie elementów z początku lub końca

```
slice1 := []int{1, 2, 3, 4}

slice2 := slice1[:3]
fmt.Println(slice2)
// => [1 2 3]

slice3 := slice1[1:]
fmt.Println(slice3)
// => [2 3 4]
```

Arrays & Slices

Usuwanie elementów ze środka

```
slice1 := []int{1, 2, 3, 4, 5}

slice2 := append(slice1[:2], slice1[3:]...)
fmt.Println(slice2)
// => [1 2 4 5]
```



Arrays & Slices

Iteracja

```
for _, v := range []string{"a", "b", "c"} {
    fmt.Println(v)
}
// => a
// => b
// => c
```

Maps

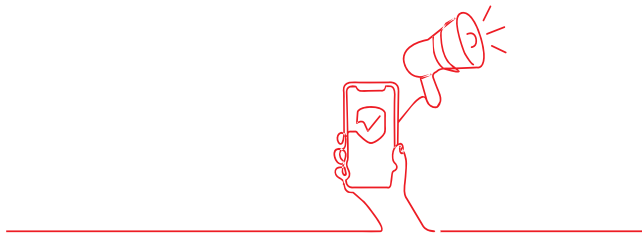
```go
var temp map[string]int
fmt.Println(temp)
// => map[]

fmt.Println(temp == nil)
// => true
```

- Mapa przechowuje dane w postaci klucz - wartość.
- Klucze są unikalne.
- Domyślną wartością jest `nil`.

Maps

```go
var temp map[string]int

fmt.Println(temp["2023-01-31"])
// => 0

temp["2023-01-31"] = 2
// panic: assignment to entry in nil map
```

- Odczyt z niezainicjalizowanej mapy zwraca domyślną wartość.
- Zapis kończy się krytycznym błędem.



Maps

```go
temp := map[string]int{"2022-01-31": 2}
fmt.Println(len(temp))
// => 1

empty := make(map[int]int, 10)
fmt.Println(len(empty))
// => 0
fmt.Println(empty == nil)
// => false
```

- Mapę możemy inicjalizować na dwa sposoby: za pomocą {} oraz make()
- “Długość” mapy to liczba jej elementów.

Maps

```go
temp := map[string]int{"2022-01-31": 2}

fmt.Println(temp["does not exist"])
// => 0

value, ok := temp["2022-02-01"]
if ok {
    fmt.Println("exists:", value)
} else {
    fmt.Println("does not exist")
}
// => does not exist
```

- Mapa zawsze zwraca wartość - jeśli klucz nie istnieje, to zwracana jest domyślna wartość.
- Drugą (opcjonalną) zwracaną wartością jest `bool` informujący czy klucz istnieje.

Maps

```go
temp := map[string]int{
    "2022-01-31": 2,
    "2022-02-01": 4,
}

delete(temp, "2022-01-31")
fmt.Println(temp)
// => map[2022-02-01:4]
```

→ `delete` usuwa wartość “w miejscu”.

Maps

```go
iter := map[int]string{1: "a", 2: "b", 3: "c"}

for key, value := range iter {
    fmt.Println(key, "-", value)
}
// => 3 - c
// => 1 - a
// => 2 - b
```

- Po mapach można iterować - zwracany jest klucz lub klucz i wartość.
- Kolejność jest losowa.

Structs

```
type car struct {
    model                string
    engineCapacity       int
    automaticTransmission bool
}

type empty struct{}
```

- Struktura to zbiór atrybutów.
- Struktura bez atrybutów jest dozwolona.

Structs

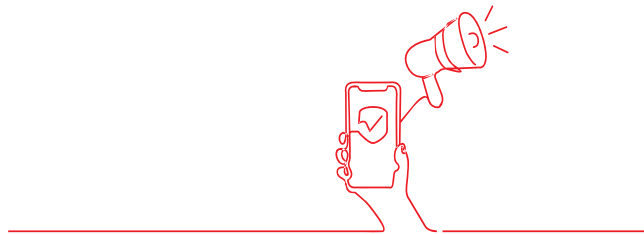
```
ford := car{
    model: "Focus",
    engineCapacity: 1560,
}
fmt.Println(ford)
// => {Focus 1560 false}

fmt.Println(car{})
// => { 0 false}
```

- Przy inicjalizacji struktury, niepodane atrybuty przyjmują domyślną wartość.
- Dozwolone jest niepodanie żadnych atrybutów.

Structs

```
var ford car
fmt.Println(ford)
// => { 0 false}

ford.model = "Focus"
fmt.Println(ford)
// => {Focus 0 false}

fmt.Println(ford.engineCapacity)
// => 0
```

→ Do atrybutów można odnosić się “po kropce”.



Structs

```go
empty := struct{}{}
fmt.Println(unsafe.Sizeof(empty))
// => 0

boolean := false
fmt.Println(unsafe.Sizeof(boolean))
// => 1
set := map[int]struct{}{1: {}, 2: {}, 3: {}}
if _, contains := set[5]; !contains {
    fmt.Println("not in a set")
}
// => not in a set
```

- Puste struktury (nie mające atrybutów) nie zajmują miejsca.
- Można ten fakt wykorzystać do zaimplementowania setu przy użyciu mapy.

Pointers

```go
var pointer *int
fmt.Println(pointer)
// => <nil>

foo := 10

pointer = &foo
fmt.Println(pointer)
// => 0xc0000ac010
```

- Domyślna wartość to `nil`.
- Każdy typ danych ma własny typ wskaźnika o tej samej nazwie co typ, poprzedzony znakiem *
- Wskaźnik przechowuje adres pamięci, gdzie znajduje się wartość.
- Adres zmiennej można uzyskać używając &.

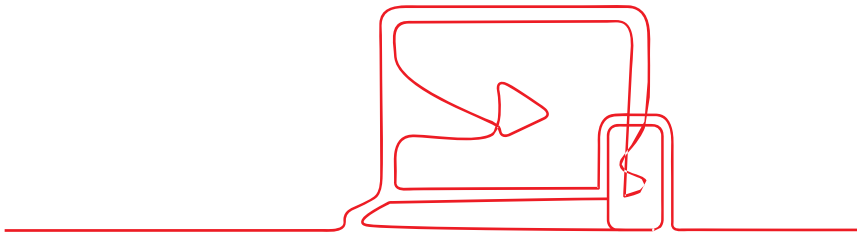
Pointers

```
foo := 10
pointer := &foo
fmt.Println(*pointer)
// => 10

foo = 20
fmt.Println(*pointer)
// => 20
```

→ Wartość kryjącą się pod danym adresem można uzyskać używając *.

Po co wskaźniki?

- Umożliwiają przekazanie wartości bez kopiowania.
- Pozwalają odróżnić brak wartości (`nil`) od wartości domyślnej.



Ciekawostka

```
m := make(map[float64]string)

m[math.NaN()] = "I'm"
m[math.Sqrt(-1)] = "indestructible!"

for k := range m {
    delete(m, k)
}

fmt.Printf("length: %d, contents: %#v\n", len(m), m)
// => length: 2, contents: map[float64]string{NaN:"indestructible!", NaN:"I'm"}
```

Ciekawostka

```
m := make(map[float64]string)

m[math.NaN()] = "No longer"
m[math.Sqrt(-1)] = "indestructible!"

clear(m)

fmt.Printf("length: %d, contents: %#v\n", len(m), m)
// => length: 0, contents: map[float64]string{}
```

Zadanie do przećwiczenia materiału

→ github.com/grupawp/akademia-programowania-2/Golang/zadania/academy