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

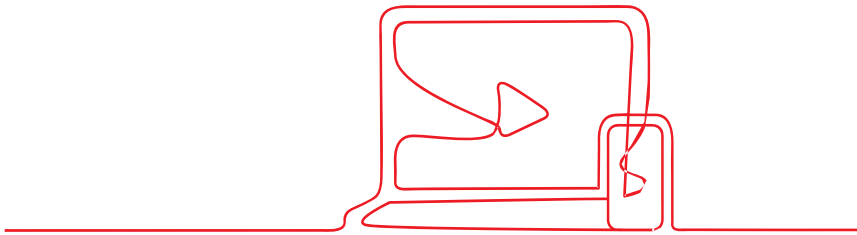
```go
var emptyVariable string
var unused string

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

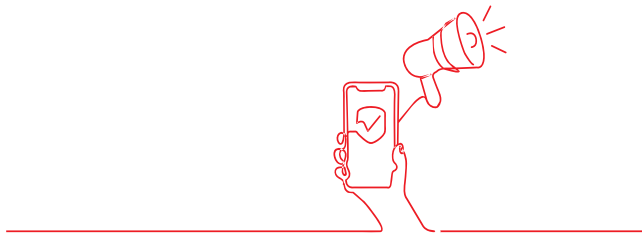
```go
var signedInt int8
fmt.Println(signedInt - 1)
// => -1

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

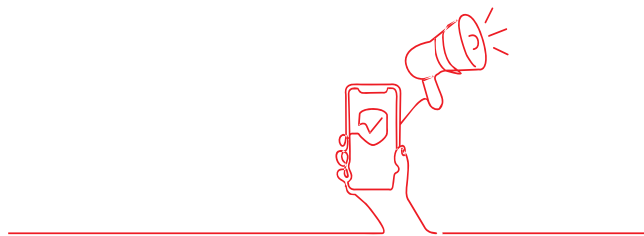
```go
if pi := math.Pi; pi < 3 {
    fmt.Println("?")
} else if pi >= 3 {
    fmt.Printf("%.2f\n", pi)
}
// => 3.14
```

- 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.

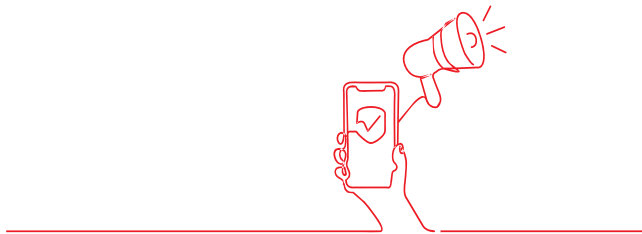
For

```
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

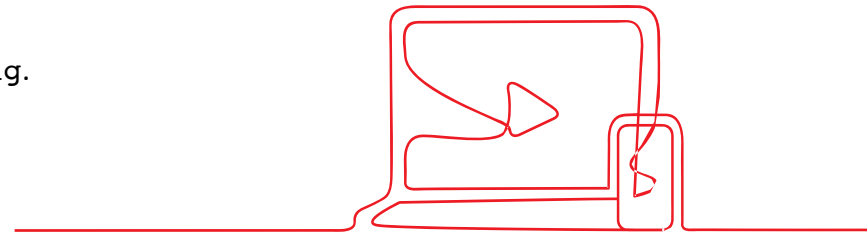
```
length := 0
for range 10 {
    length++
}
fmt.Println(length)
// => 10
```

→ 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

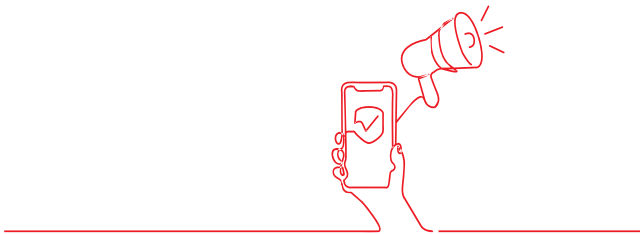
```go
var array [10]int
fmt.Println(array)
// => [0 0 0 0 0 0 0 0 0 0]

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

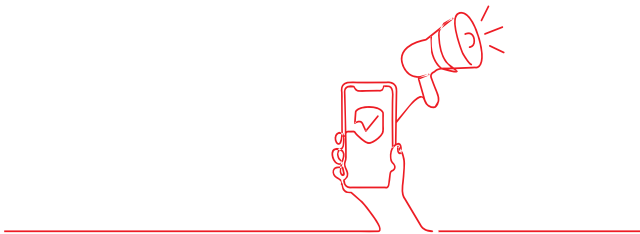
Arrays & Slices

```
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

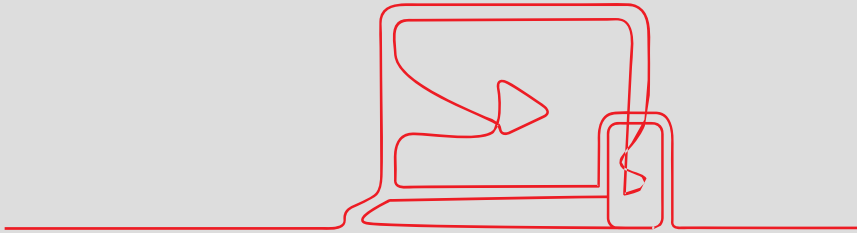
Usuwanie elementów z początku lub końca

```
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

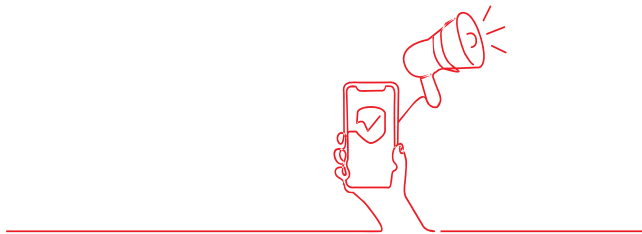
```go
var temp map[string]int
fmt.Println(temp)
// => map[]

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

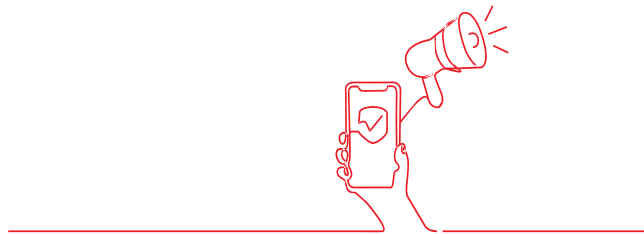
```
ford := car{
    model: "Focus",
    engineCapacity: 1560,
}
fmt.Println(ford)
// => {Focus 1560 false}

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 &.

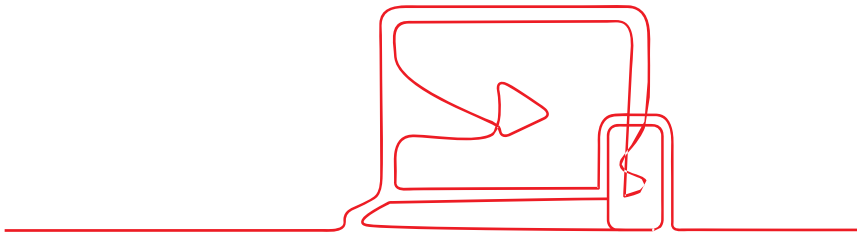
Pointers

```
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

Multiple return values

- Deklaracja funkcji może zawierac kilka zwracanych wartości.
- Wartości mogą być różnego typu.
- playground

```
func addSubDiv(a, b int) (int, int, float64) {
    add := a + b
    sub := a - b
    div := float64(a) / float64(b)
    return add, sub, div // (int, int, float64)
}

func main() {
    add, sub, div := addSubDiv(11, 5)

    fmt.Printf("Add: %d, Sub: %d, Div: %.2f\n", add, sub, div)
}
```

Named return values

- Zwracane wartości mogą mieć swoje nazwy.

```
func addSubDiv(a, b int) (add int, sub int, div float64) {
    add = a + b
    sub = a - b
    div = float64(a) / float64(b)

    return
}

func main() {
    add, sub, div := addSubDiv(11, 5)

    fmt.Printf("Add: %d, Sub: %d, Div: %.2f\n", add, sub, div)
}
```

Pointer receiver

- Metoda, jako działanie na rzecz struktury
- playground

```
type Adder struct {
    Sum int
    Sub int
}

func (a *Adder) add(x, y int) {
    a.Sum = x + y
}

func sub(a *Adder, x, y int) {
    a.Sub = x - y
}

func main() {
    adder := Adder{}

    adder.add(2, 1)
    sub(&adder, 2, 1)

    fmt.Printf("Sum: %d, Sub: %d\n", adder.Sum, adder.Sub)
}
```

Value receiver

- Receiver jako “wartość”.
- playground

```
type Adder struct {
    Sum int
    Sub int
}

func (a Adder) add(x, y int) {
    a.Sum = x + y
}

func sub(a Adder, x, y int) Adder {
    a.Sub = x - y

    return a
}

func main() {
    adder := Adder{}

    adder.add(2, 1)
    adder = sub(adder, 2, 1)

    fmt.Printf("Sum: %d, Sub: %d\n", adder.Sum, adder.Sub)
}
```

New keyword

- Tworzenie struktur za pomocą słowa kluczowego `new`

```
package main

import "fmt"

type Foo struct {
    Bar string
}

func main() {
    foo := new(Foo)
    foo.Bar = "Baz"
    fmt.Printf("foo: %+v = %+v\n", foo, *foo)
    // output: foo: &{Bar:Baz} = {Bar:Baz}
}
```

Make keyword

- Tworzenie tablic, kanałów lub map przy pomocy `make`

```
package main

import "fmt"

func main() {

    a := make([]int, 10) // stwórz tablicę o pojemności 10
    fmt.Printf("len: %d, cap: %d, val: %v\n", len(a), cap(a), a)
    // output: len: 10, cap: 10, val: [0 0 0 0 0 0 0 0 0 0]

    ch := make(chan int, 4) // swtórz kanał o pojemności 4
    ch <- 1
    fmt.Printf("len: %d, cap: %d, val: %v\n", len(ch), cap(ch), ch)
    // output: len: 1, cap: 4, val: 0xc000110000

    m := make(map[int]int) // swtórz mapę o pojemności ?
    fmt.Printf("len: %d, val: %v\n", len(m), m)
    // output: len: 0, val: map[]
}
```

INTERFEJSY

Pakiet `draw` rysujący obiekty na ekranie

```
package draw

type Screen struct {
}

func (s *Screen) DrawRect(x, y, width, height int) {}

func (s *Screen) DrawCircle(x, y, r int) {}
```

Wykorzystanie i testowanie pakietu `draw`

```go
package engine
import "example/draw"

func Refresh(s *draw.Screen) {
    // ...
    s.DrawRect(100,100,25,10)
}
```

- W teście jednostkowym jesteśmy zmuszeni wykorzystać typ draw.Screen
- Nie chcemy testować pakietu `draw` tylko metodę `Refresh`
- Chcemy aby metoda `Refresh` przyjęła w parametrze typ, nad którego implementacją mamy kontrolę podczas testu

Duck Typing

Duck typing – rozpoznawanie typu obiektu nie na podstawie deklaracji, ale przez badanie metod udostępnionych przez obiekt. Technika ta wywodzi się z powiedzenia: „jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką”.

```
type RoboDuck struct {}
func (d *RoboDuck) Quack() {}
func (d *RoboDuck) Walk() {}
```

```
type Duck struct {}
func (d *Duck) Quack() {}
func (d *Duck) Walk() {}
```

Interfejs

- Interfejs składa się z sygnatur metod
- Sygnatura to nazwa metody, lista parametrów wejściowych i typów wyjściowych (o ile występują)

```
type Drawer interface {
    DrawRect(int, int, int, int)
    DrawCircle(int, int, int)
}
```

Interfejs

- Uwaga! To jest antypattern!
- Interfejs powinien zostać zdefiniowany po stronie “klienta”, czyli w pakiecie `engine`

```
package draw

type Drawer interface {
    DrawRect(int, int, int, int)
    DrawCircle(int, int, int)
}

type Screen struct {
}

func (s *Screen) DrawRect(x, y, width, height int) {}
func (s *Screen) DrawCircle(x, y, r int) {}
```

```
package engine
import "example/draw"

func Refresh(s draw.Drawer) {
    s.DrawRect(100,100,25,10)
}
```

Interfejs

Dlaczego definicja interfejsu powinna znajdować się po stronie “klienta”?

- Zmiana pakietu `draw` na inny, który nie zawiera metody `DrawCircle`, wymusza zmiany we wszystkich metodach korzystających z `draw.Drawer`
- Importując interfejs `draw.Drawer` nie mamy wpływu na zawartość interfejsu (listę sygnatur metod)

Jak to zrobić lepiej?

```
package engine
import "example/draw"

type Drawer interface {
    DrawRect(int, int, int, int)
    DrawCircle(int, int, int)
}

func Refresh(s Drawer) {
    s.DrawRect(100,100,25,10)
}
```

Nie trzeba korzystać ze wszystkich metod!

```go
package engine
import "example/draw"

// type Drawer interface {
//     DrawRect(int, int, int, int)
//     DrawCircle(int, int, int)
// }

type Rectangler interface {
    DrawRect(int, int, int, int)
}

func Refresh(r Rectangler) {
    r.DrawRect(100,100,25,10)
}
```

Pusty interface (any)

- Szczególnym przypadkiem jest typ `interface{}` lub jego alias `any`
- Dowolny typ spełnia interfejs bez metod jednak nie powinien być nadużywany

Best Practices

- Przyjmuj interfejsy, zwracaj struktury
- Interfejs spełnia jedną logikę związaną z jego nazwą
- Nazwa interfejsu zazwyczaj jest rzeczownikiem kończącym się na -er

Przykłady interfejsów w bibliotece standardowej Go

```go
type error interface {
    Error() string
}
```

```go
package main

import "fmt"

type myErr struct {
    msg  string
    code int
}

func (e *myErr) Error() string {
    return fmt.Sprintf("%s, code %d", e.msg, e.code)
}
```

```go
func myFunc() *myErr {
    return &myErr{
        msg:  "not found",
        code: 404,
    }
}

func myService() error {
   e := myFunc()
   return fmt.Errorf("myFunc returned error: %v",e)
}

func main() {
    if err := myService(); err != nil {
        fmt.Println(err)
        return
    }
}

// output:
// myFunc returned error: not found, code 404
```

Przykłady interfejsów w bibliotece standardowej Go

Pakiet fmt

```
type Stringer interface {
    String() string
}
```

```
package main
import "fmt"

type Box struct {
    x, y, width, height int
}

// func (b *Box) String() string {
//     return fmt.Sprintf("X: %d, Y: %d, Width: %d, Height: %d",
//                        b.x, b.y, b.width, b.height)
//}

func main() {
        b := new(Box)
        fmt.Println(b)
}

// output:
// &{0 0 0 0}

// output z metodą String()
// X: 0, Y: 0, Width: 0, Height: 0
```

Zapoznaj się z innymi interfejsami w stdlib

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type ReadCloser interface {
    Reader
    Closer
}

type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

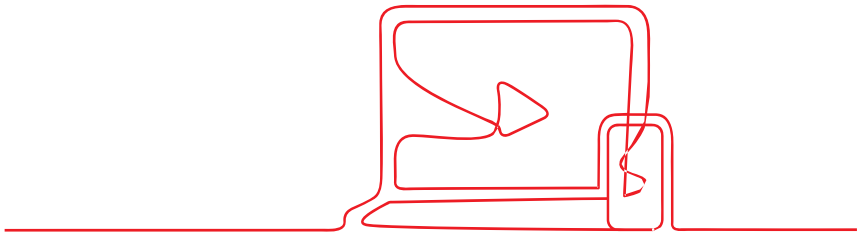
POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

Podstawy Go

Konwersja typów

- Go jest językiem silnie typowanym - zawsze kontroluje typy używanych zmiennych
- Jeśli chcemy użyć wartości w innym kontekście, musimy jawnie skonwertować typ
- wyrażenie T(v) konwertuje wartość v do typu T

```
var i int = 2
var f float64
f = float64(i) // T(v)
```



Inferencja typów

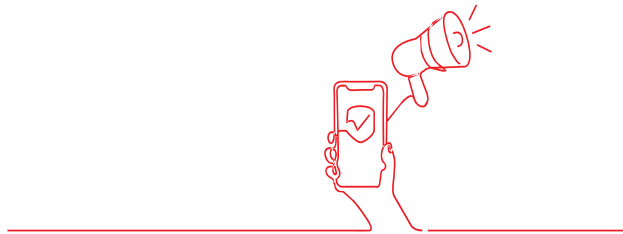
- W niektórych przypadkach kompilator może wywnioskować, jakiego typu zmienną definiujemy

- Tak będzie, jeśli odwołamy się do już istniejącej zmiennej:

```
var x int
var z = x
```

- Możemy też podać stałą - odpowiedni typ zostanie dobrany automatycznie:

```
i := 1337   // int
f := 13.37  // float64
```



Asercja typów

- Za interfejsem stoi konkretny typ - nie można jednak odwołać się do niego bezpośrednio

- Asercja typu pozwala “wyłuskać” wartość `t`, typu `T` z wartości `i`:

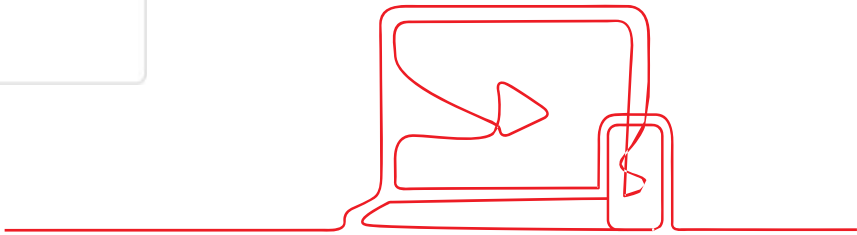
```
t := i.(T)

var i interface{} = "jestem stringiem z Koniakowa"
s := i.(string)
```

- Uzyskamy zmienną `s` typu string

Asercja typów, cd.

- Jeśli wartość interfejsu nie jest typu T, program zakończy się z błędem
- By tego uniknąć, możemy skorzystać z formy zwracającej dwie wartości
- Pierwsza to - jak poprzednio - zmienna konkretnego typu, druga - boolean - wskazuje na możliwość wykonania operacji

```
t, ok := i.(T)

s, ok := i.(string)  // s = "jestem... ", ok = true
f, ok := i.(float64) // f = 0, ok = false
```



Type switch

- Specjalna konstrukcja `switch` umożliwia proste sprawdzenie kilku typów mogących się kryć za interfejsem
- Tutaj w miejsce nazwy typu trafia słowo `type`, a zwracane są nie wartości, a nazwy typów:

```
switch x := i.(type) {
    case int:
        ...
    case string:
        ...
    default:
        ...
}
```

Go Playground - przykład

- Zmienna `x` będzie zawierać wartość konkretnego typu (poza `default`, gdzie `x` będzie po prostu równe `i`)

Funkcje jako wartości

- W Go funkcje mogą być traktowane również jako wartości
- Mogą być przypisywane, zwracane, używane jako argumenty innych funkcji

```
func razyDwa(x int) int {
    return 2 * x
}

func uruchom(fn func(x int) int, x int) int {
    return fn(x)
}
...
f := razyDwa
fmt.Println(f(5))
fmt.Println(uruchom(f, 5))
```

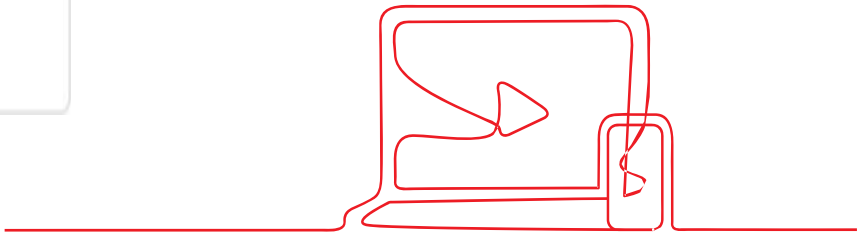
Funkcje anonimowe

- Nie wszystkie funkcje muszą mieć nazwę - możemy też definiować równie użyteczne funkcje anonimowe

```
fmt.Println(func(x int) int {return 2 * x}(5))
```

- Funkcja może też zwracać funkcje anonimowe:

```
func dajWitacz() func(string) {
    return func(imię string) {
        fmt.Println("Cześć,", imię)
    }
}
...
x := dajWitacz()
x("Wojtek")
```



Funkcje - domknięcia

- Tworzone w ciele innej funkcji funkcje anonimowe zachowują dostęp do definiowanych w “rodzicu” zmiennych
- Dane te nie są niszczone po zakończeniu rodzica, można z nich dalej korzystać

```
func plusJeden() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}
...
x := plusJeden()
fmt.Println(x()) // 1
fmt.Println(x()) // 2
```

- `plusJeden` zwróciło funkcję, kolejne jej wywołania wskazują, że zmienna `i` jest dalej dostępna

Go Playground - uwaga, do poprawki!

Defer

- `defer` daje możliwość “zamówienia” wykonania kodu przed wyjściem z aktualnie wykonującej się funkcji

```
func main() {
    defer fmt.Println("Do zobaczenia!")

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

- Wskazana po `defer` funkcja wykona się niezależnie od wybranej ścieżki wykonania

- Bardzo często wykorzystywany np. do zwalniania pobranych zasobów, zamykania połączeń

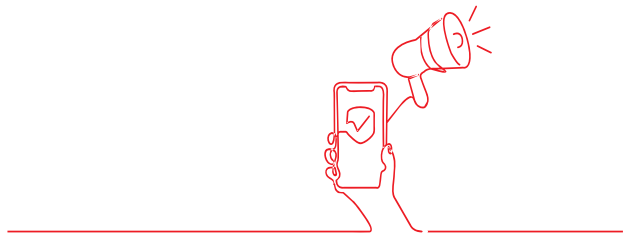
Defer

- `defer` przekazuje parametry w momencie definicji, nie późniejszego wywołania
- Można korzystać z `defer` wielokrotnie
- Wywołania umieszczane są na stosie, wykonywane w odwrotnej kolejności
- Kod może zmieniać wartość nazwanych wyników

```
func x() (ret int) {
    for i := 0; i < 5; i++ {
        defer func() { ret++; fmt.Println(ret) }()
    }
    return 5
}
```

Go Playground 1

Go Playground 2



POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

PAKIETY, EKSPORTY, GO MOD

Go modules

- Moduł jest to zbiór pakietów/zależności wchodzących w skład projektu
- Jego definicja znajduje się w pliku `go.mod`

Tworzenie nowego modułu

1. Tworzymy folder dla naszego projektu oraz przechodzimy do jego lokalizacji

2.
```
$ mkdir pjatk_project
$ cd pjatk_project
```

2. Korzystając z narzędzia `go mod init` tworzymy plik `go.mod` jako argument przekazując nazwę modułu

```
$ go mod init pjatk_project
```

W folderze projektu został wygenerowany plik `go.mod`

```
pjatk_project/
└── go.mod
```

```
module pjatk_project

go 1.19
```

W aktualnej formie plik zawiera informacje: - nazwa modułu - wersja Go

Dodawanie zewnętrznych zależności do projektu

- Chcemy aby nasz program po uruchomieniu wyświetlał unikalny identyfikator UUID
- W tym celu wykorzystamy bibliotekę dostarczaną przez Google https://github.com/google/uuid

Instalujemy zależność z pomocą narzędzia `go get`

```
$ go get github.com/google/uuid
```

```
module pjatk_project

go 1.19

require github.com/google/uuid v1.3.0 // indirect
```

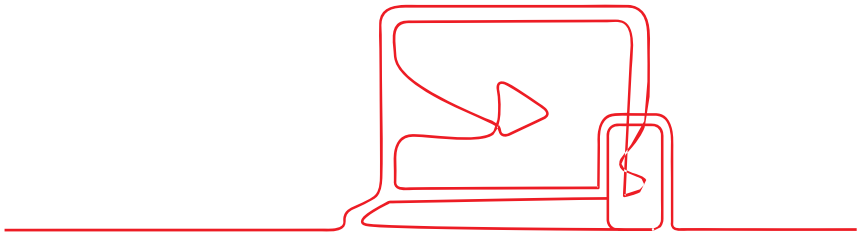
- w pliku `go.mod` została dodana nowa sekcja `require` opisująca dodaną zależność oraz jej wersję
- oznaczenie `//indirect` informuje o tym, że zainstalowana zależność nie jest jawnie wykorzystana w naszym kodzie

Dodawanie zewnętrznych zależności do projektu

- W projekcie pojawił się również kolejny plik `go.sum`

```
pjatk_project/
├── go.mod
└── go.sum

github.com/google/uuid v1.3.0 h1:t6JiXgmwXMjEs8VusXIJk2BXHsn+wx8BZdTaoZ5fu7I=
github.com/google/uuid v1.3.0/go.mod h1:TIyPZe4MgqvfeYDBFedMoGGpEw/LqOeaOT+nhxU+yHo=
```

- Plik ten zawiera informacje takie jak nazwa, wersja oraz hash zależności.



Dodawanie zewnętrznych zależności do projektu

Teraz możemy wykorzystać bibliotekę UUID [playground]

```
package main

import (
    "fmt"
    "github.com/google/uuid"
)

func main() {
    uniqueToken, _ := uuid.NewRandom()
    fmt.Printf("Twój unikalny token to: %s", uniqueToken)
}
```

```
$ go run main.go

Twój unikalny token to: 674c2952-0704-42f2-86de-7f542ab8d576
```

Zmiany zależności

W trakcie pisania programu możemy chcieć zmodyfikować, lub usunąć niektóre biblioteki. Aby zachować spójność zależności w plikach `go.mod` i `go.sum` Go dostarcza operację `mod tidy` która doda, zmodyfikuje lub usunie je automatycznie.

Przykładowo chcemy by nasz unikalny token był generowany przez bibliotekę *https://github.com/thanhpk/randstr*

```
$ go get github.com/thanhpk/randstr
```

po zainstalowaniu zależności dokonujemy zmian w kodzie playground

```go
package main

import (
    "fmt"
    "github.com/thanhpk/randstr"
)

func main() {
    uniqueToken := randstr.String(16)
    fmt.Printf("Twój unikalny token to: %s", uniqueToken)
}
```

Zmiany zależności

wykonujemy polecenie

```
$ go mod tidy
```

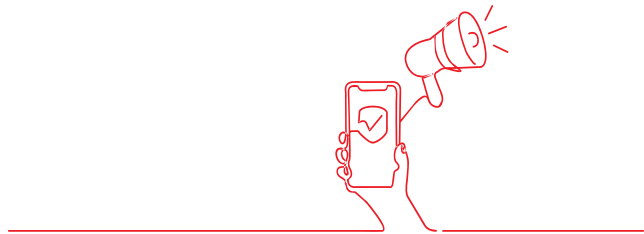
W jego wyniku pliki zostały zaktualizowane i nieużywana zależność UUID została usunięta

```
module pjatk_project

go 1.19

require github.com/thanhpk/randstr v1.0.4
github.com/thanhpk/randstr v1.0.4 h1:IN78qu/bR+My+gHCvMEXhR/i5oriVHcTB/BJJIRTsNo=
github.com/thanhpk/randstr v1.0.4/go.mod
h1:M/H2P1eNLZzlDwAzpkkkUvoyNNMbzRGhESZuEQk3r0U=
```

Tworzenie własnych pakietów i eksportowanie nazw

- Pakiety są paczkami kodu źródłowego
- Służą do grupowania i udostępniania funkcji, typów i zmiennych w celu ich wielokrotnego użycia
- Każdy plik z kodem źródłowym musi przynależeć do jakiegoś pakietu

Chcielibyśmy wydzielić funkcjonalność generowania tokenu do oddzielnej paczki, by nie zawierać całej logiki w pliku `main.go`.



Tworzenie własnych pakietów i eksportowanie nazw

Na początku tworzymy nowy folder `token` w naszym projekcie oraz dodajemy plik `generator.go`

```
pjatk_project/
├── go.mod
├── go.sum
├── main.go
└── token
        └── generator.go
```

Naszą paczkę nazwiemy token i będzie ona odpowiedzialna za operacje na tokenach. W pliku `generator.go` definujemy nową paczkę umieszczając na początku pliku:

```go
package token
```

Tworzenie własnych pakietów i eksportowanie nazw

Przenieśmy logikę generowania tokenu z pliku `main.go` do nowej funkcji w `generator.go`

```go
package token

import "github.com/thanhpk/randstr"

func generate(len int) string {
    return randstr.String(len)
}
```

Tworzenie własnych pakietów i eksportowanie nazw

Następnie w pliku `main.go` dokonajmy modyfikacji by wykorzystać nasz nowy pakiet `token` oraz funkcję `generate`

```
    package main

    import (
        "fmt"
        "pjatk_project/token"
    )

    func main() {
        uniqueToken := token.generate(16)
            fmt.Printf("Twój unikalny token to: %s", uniqueToken)
    }
```

Tworzenie własnych pakietów i eksportowanie nazw

Spróbujmy teraz uruchomić nasz projekt

```
$ go run main.go
    ./main.go:9:23: undefined: token.generate
```

W konsoli pojawił się błąd, ponieważ nasza funkcja `generate` nie została eksportowana i nie jest widoczna nigdzie poza pakietem `token`.

Aby naprawić błąd musimy wykonać eksport. Jedyne co należy zrobić to zmienić nazwę funkcji w `generator.go` tak by zaczynała się z wielkiej litery:

```
func Generate(len int) string
```

Teraz zedytujmy wywołanie funkcji w pliku `main.go`

```
uniqueToken := token.Generate(16)
```

Sukces! Właśnie stworzyliśmy naszą pierwszą paczkę i eksportowaliśmy jej funkcję!

Tworzenie własnych pakietów i eksportowanie nazw

W Go nazwy typów, funkcji, zmiennych, stałych itd. mogą zostać eksportowane poprzez zastosowanie w nazwie wielkiej litery na początku. Takie nazwy są `publiczne` i dostępne również poza pakietem w którym zostały zadeklarowane. Natomiast nazwy zaczynające się małą literą są `prywatne` i dostęp do nich jest ograniczany tylko z poziomu tej samej paczki w której zostały utworzone.

Podsumowanie

- Moduły służą do zarządzania zależnościami w projekcie
- Go udostępnia wbudowane narzędzia (więcej informacji w dokumentacji: https://go.dev/ref/mod)
  - `go mod init <nazwa>`- tworzenie modułu
  - `go get <zależność>` - instalowanie biblioteki
  - `go mod tidy` - aktualizacja zależności
- Pakiety służą do grupowania kodu źródłowego
- By korzystać z zasobów paczki w innych miejscach aplikacji (innych paczkach) musimy zrobić je publiczne poprzez `eksportowanie nazw`

POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

PAKIETY, EKSPORTY, GO MOD

Go modules

- Moduł jest to zbiór pakietów/zależności wchodzących w skład projektu
- Jego definicja znajduje się w pliku `go.mod`

Tworzenie nowego modułu

1. Tworzymy folder dla naszego projektu oraz przechodzimy do jego lokalizacji

2.
```
$ mkdir pjatk_project
$ cd pjatk_project
```

2. Korzystając z narzędzia `go mod init` tworzymy plik `go.mod` jako argument przekazując nazwę modułu

```
$ go mod init pjatk_project
```

W folderze projektu został wygenerowany plik `go.mod`

```
pjatk_project/
└── go.mod
```

```
module pjatk_project

go 1.19
```

W aktualnej formie plik zawiera informacje: - nazwa modułu - wersja Go

Dodawanie zewnętrznych zależności do projektu

- Chcemy aby nasz program po uruchomieniu wyświetlał unikalny identyfikator UUID
- W tym celu wykorzystamy bibliotekę dostarczaną przez Google https://github.com/google/uuid

Instalujemy zależność z pomocą narzędzia `go get`

```
$ go get github.com/google/uuid
```

```
module pjatk_project

go 1.19

require github.com/google/uuid v1.3.0 // indirect
```

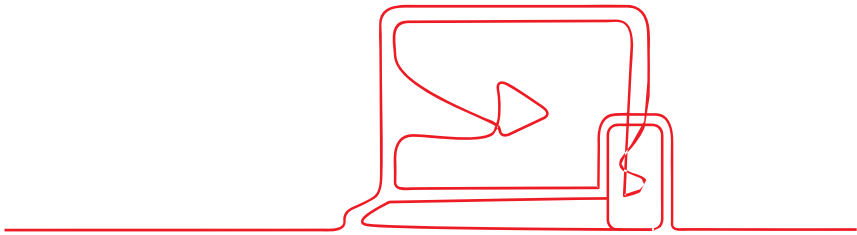
- w pliku `go.mod` została dodana nowa sekcja `require` opisująca dodaną zależność oraz jej wersję
- oznaczenie `//indirect` informuje o tym, że zainstalowana zależność nie jest jawnie wykorzystana w naszym kodzie

Dodawanie zewnętrznych zależności do projektu

- W projekcie pojawił się również kolejny plik `go.sum`

```
pjatk_project/
├── go.mod
└── go.sum

github.com/google/uuid v1.3.0 h1:t6JiXgmwXMjEs8VusXIJk2BXHsn+wx8BZdTaoZ5fu7I=
github.com/google/uuid v1.3.0/go.mod h1:TIyPZe4MgqvfeYDBFedMoGGpEw/LqOeaOT+nhxU+yHo=
```

- Plik ten zawiera informacje takie jak nazwa, wersja oraz hash zależności.



Dodawanie zewnętrznych zależności do projektu

Teraz możemy wykorzystać bibliotekę UUID [playground]

```
package main

import (
    "fmt"
    "github.com/google/uuid"
)

func main() {
    uniqueToken, _ := uuid.NewRandom()
    fmt.Printf("Twój unikalny token to: %s", uniqueToken)
}
```

```
$ go run main.go

Twój unikalny token to: 674c2952-0704-42f2-86de-7f542ab8d576
```

Zmiany zależności

W trakcie pisania programu możemy chcieć zmodyfikować, lub usunąć niektóre biblioteki. Aby zachować spójność zależności w plikach `go.mod` i `go.sum` Go dostarcza operację `mod tidy` która doda, zmodyfikuje lub usunie je automatycznie.

Przykładowo chcemy by nasz unikalny token był generowany przez bibliotekę *https://github.com/thanhpk/randstr*

```
$ go get github.com/thanhpk/randstr
```

po zainstalowaniu zależności dokonujemy zmian w kodzie playground

```go
package main

import (
    "fmt"
    "github.com/thanhpk/randstr"
)

func main() {
    uniqueToken := randstr.String(16)
    fmt.Printf("Twój unikalny token to: %s", uniqueToken)
}
```

Zmiany zależności

wykonujemy polecenie

```
$ go mod tidy
```

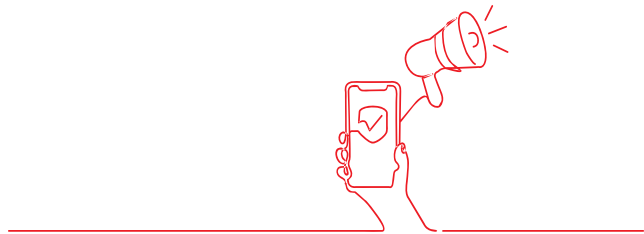
W jego wyniku pliki zostały zaktualizowane i nieużywana zależność UUID została usunięta

```
module pjatk_project

go 1.19

require github.com/thanhpk/randstr v1.0.4
github.com/thanhpk/randstr v1.0.4 h1:IN78qu/bR+My+gHCvMEXhR/i5oriVHcTB/BJJIRTsNo=
github.com/thanhpk/randstr v1.0.4/go.mod
h1:M/H2P1eNLZzlDwAzpkkkUvoyNNMbzRGhESZuEQk3r0U=
```

Tworzenie własnych pakietów i eksportowanie nazw

- Pakiety są paczkami kodu źródłowego
- Służą do grupowania i udostępniania funkcji, typów i zmiennych w celu ich wielokrotnego użycia
- Każdy plik z kodem źródłowym musi przynależeć do jakiegoś pakietu

Chcielibyśmy wydzielić funkcjonalność generowania tokenu do oddzielnej paczki, by nie zawierać całej logiki w pliku `main.go`.



Tworzenie własnych pakietów i eksportowanie nazw

Na początku tworzymy nowy folder `token` w naszym projekcie oraz dodajemy plik `generator.go`

```
pjatk_project/
├── go.mod
├── go.sum
├── main.go
└── token
        └── generator.go
```

Naszą paczkę nazwiemy token i będzie ona odpowiedzialna za operacje na tokenach. W pliku `generator.go` definujemy nową paczkę umieszczając na początku pliku:

```go
package token
```

Tworzenie własnych pakietów i eksportowanie nazw

Przenieśmy logikę generowania tokenu z pliku `main.go` do nowej funkcji w `generator.go`

```go
package token

import "github.com/thanhpk/randstr"

func generate(len int) string {
    return randstr.String(len)
}
```

Tworzenie własnych pakietów i eksportowanie nazw

Następnie w pliku `main.go` dokonajmy modyfikacji by wykorzystać nasz nowy pakiet `token` oraz funkcję `generate`

```
    package main

    import (
        "fmt"
        "pjatk_project/token"
    )

    func main() {
        uniqueToken := token.generate(16)
            fmt.Printf("Twój unikalny token to: %s", uniqueToken)
    }
```

Tworzenie własnych pakietów i eksportowanie nazw

Spróbujmy teraz uruchomić nasz projekt

```
$ go run main.go
    ./main.go:9:23: undefined: token.generate
```

W konsoli pojawił się błąd, ponieważ nasza funkcja `generate` nie została eksportowana i nie jest widoczna nigdzie poza pakietem `token`.

Aby naprawić błąd musimy wykonać eksport. Jedyne co należy zrobić to zmienić nazwę funkcji w `generator.go` tak by zaczynała się z wielkiej litery:

```
func Generate(len int) string
```

Teraz zedytujmy wywołanie funkcji w pliku `main.go`

```
uniqueToken := token.Generate(16)
```

Sukces! Właśnie stworzyliśmy naszą pierwszą paczkę i eksportowaliśmy jej funkcję!

Tworzenie własnych pakietów i eksportowanie nazw

W Go nazwy typów, funkcji, zmiennych, stałych itd. mogą zostać eksportowane poprzez zastosowanie w nazwie wielkiej litery na początku. Takie nazwy są `publiczne` i dostępne również poza pakietem w którym zostały zadeklarowane. Natomiast nazwy zaczynające się małą literą są `prywatne` i dostęp do nich jest ograniczany tylko z poziomu tej samej paczki w której zostały utworzone.

Podsumowanie

- Moduły służą do zarządzania zależnościami w projekcie
- Go udostępnia wbudowane narzędzia (więcej informacji w dokumentacji: https://go.dev/ref/mod)
  - `go mod init <nazwa>`- tworzenie modułu
  - `go get <zależność>` - instalowanie biblioteki
  - `go mod tidy` - aktualizacja zależności
- Pakiety służą do grupowania kodu źródłowego
- By korzystać z zasobów paczki w innych miejscach aplikacji (innych paczkach) musimy zrobić je publiczne poprzez `eksportowanie nazw`

POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

Biblioteka standardowa

Opis ogólny

- Biblioteka standardowa języka Go zawiera wiele użytecznych narzędzi rozwijanych przez twórców języka.
- Składa się z pakietów (packages), które należy dołączyć do własnego kodu za pomocą polecenia import, podając nazwę wymaganego pakietu.
- Pakiety biblioteki standardowej nie zawierają ścieżki tak jak ma to miejsce w pakietach tworzonych przez innych twórców.

```
import (
    "fmt"                          // pakiet z biblioteki standardowej
    "github.com/goccy/go-graphviz"  // dowolny pakiet pobrany z zewnątrz
)
```

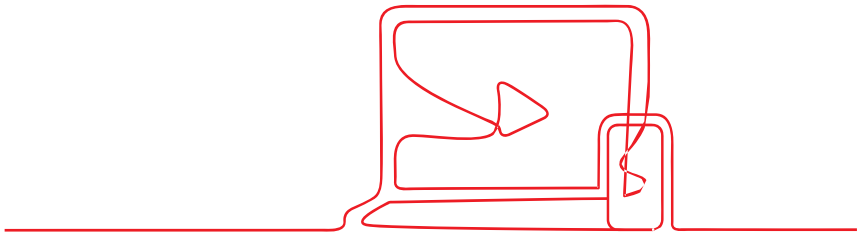
- Kompletna dokumentacja biblioteki standardowej znajduje się pod adresem: https://pkg.go.dev/std

Pakiet bytes

- Pakiet `bytes` umożliwia operacje na wycinku (slice) bajtów `[]byte`. Funkcje tego pakietu są analogiczne do funkcji pakietu `strings` który zostanie omówiony później.
- Przykładowe funkcje pakietu `bytes` to:
  - Compare(a, b []byte) int
  - Contains(b, subslice []byte) bool
  - HasPrefix(s, prefix []byte) bool
  - Index(s, sep []byte) int
  - ReplaceAll(s, old, new []byte) []byte
  - Split(s, sep []byte) [][]byte
  - ToLower(s []byte) []byte

Typ bytes.Buffer

- Jednym z częściej używanych elementów pakietu `bytes` jest typ `bytes.Buffer`, który reprezentuje bufor bajtów
- Typ implementuje wiele metod (sprawdź dokumentację https://pkg.go.dev/bytes#Buffer), warto zapamiętać dwie z nich:
  - Read(p []byte) (n int, err error)
  - Write(p []byte) (n int, err error)
- Oznacza to, że typ `bytes.Buffer` spełnia *interfejs* `io.Reader` oraz `io.Writer`



Typ bytes.Buffer

```
    b := bytes.Buffer{}
    b.Write([]byte{'H', 'e', 'l', 'l', 'o'})
    b.Write([]byte("World"))

    rdbuf := make([]byte, 5)
    _, err := b.Read(rdbuf)
    if err != nil {
        panic(err)
    }

    fmt.Println(string(rdbuf))

    _, err = b.Read(rdbuf)
    if err != nil {
        panic(err)
    }

    fmt.Println(string(rdbuf))

// Output
// Hello
// World
```

Typ bytes.Reader

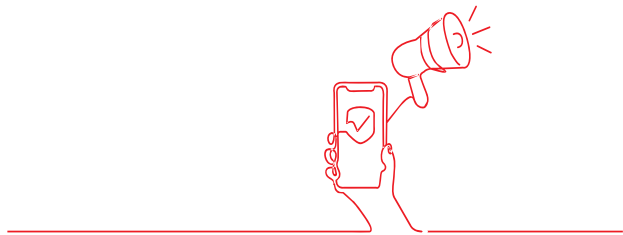
- Typ `bytes.Reader` jest również buforem bajtów
- Nie implementuje metod zapisujących dane (nie spełnia *interfejsu* `io.Writer`)
- Implementuje metodę `Read`, a więc spełnia *interfejs* `io.Reader`
- Posiada metodę `Seek` umożliwiającą ustawienie offsetu w buforze
- Zapoznaj się z dokumentacją pakietu `bytes` https://pkg.go.dev/bytes

Pakiet context

- Pakiet `context` definiuje typ `Context`
- `context.Context` przechowuje informacje sterujące elementami aplikacji do których został przekazany.
- Możesz go przekazać do funkcji, rekomendowane jest pierwsze miejsce listy parametrów.

```
func DoSomething(ctx context.Context, arg Arg) error {
}
```

- Pakiet `context` umożliwia:
  - przekazywanie parametrów klucz-wartość (ctx.WithValue)
  - zatrzymanie wykonywanego zadania w wyniku wywołania funkcji *cancel* (context.WithCancel)
  - zatrzymanie wykonywanego zadania po upływie określonego czasu (context.WithTimeout)
  - zatrzymanie wykonywanego zadania o konkretnym czasie (context.WithDeadline)



Pakiet context

- Aby utworzyć pusty context należy wykonać:

```
ctx := context.Background()
```

- Pusty context można utworzyć również za pomocą `context.TODO()`, jest to analogiczne do `context.Background()`
- Użyj `context.TODO()` jeśli nie masz pewności, który model obsługi contextu będzie odpowiedni w danym miejscu i spodziewasz się zmian w przyszłości.

```
ctx := context.TODO()
```

Pakiet context

- `ctx.WithValue` umożliwia przekazanie danych *klucz-wartość* do różnych elementów aplikacji.
- Dane mogą być dodawane w trakcie przekazywania contextu do kolejnych funkcji.
- Przykładem jest funkcja odbierająca request HTTP, która może dołączyć do contextu parametry połączenia np. adres IP, User-Agent itp.
- Innym przykładem jest przekazanie przez context wskaźnika do loggera, przez co nie ma potrzeby tworzenia zmiennych globalnych, a konfiguracja loggera znajdzie się w jednym miejscu.

```go
type logCtxKey string

const (
    logKey logCtxKey = "logger"
)

func main() {
    l := log.New(os.Stdout, "logger: ", log.Lshortfile)
    ctx := context.WithValue(context.Background(), logKey, l)
    DoSomething(ctx)
}

func DoSomething(ctx context.Context) {
    v := ctx.Value(logKey)

    l, ok := v.(*log.Logger)
    if !ok {
        panic("wrong type)")
    }

    l.Print("Hello, log file!")
}
```

Pakiet context

Częstym użyciem context.WithCancel jest bezpieczne zatrzymywanie gorutyn (graceful shutdown)

Po przekazaniu sygnału przez wywołanie funkcji *cancel*, gorutyna może bezpiecznie zakończyć realizowane zadania, zapisać dane do pliku lub bazy danych

Od wersji Go 1.20 dostępna jest funkcja WithCancelCause(parent Context) (ctx Context, cancel CancelCauseFunc), która działa jak WithCancel, ale umożliwia przekazanie błędu do funkcji cancel

```go
func worker(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            // Kończenie zadań, zamykanie zasobów.
            // ...
            fmt.Println("Shutdown completed.")
            return
        }
    }
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())

    go worker(ctx)

    // Tu jakaś praca do wykonania...
    time.Sleep(time.Second)

    // W pewnym momencie należy zatrzymać uruchomioną wcześniej gorutynę
    cancel()

    // Dalszy etap prac...
    time.Sleep(time.Second)
}
```

Pakiet context

- `context.WithTimeout` jest wywołaniem `context.WithDeadline` z parametrem *time.Now().Add(timeout)*
- Może być stosowany do przerywania wykonywanych requestów HTTP po upływie określonego czasu

```
    ctx, cancel := context.WithTimeout(context.Background(),
time.Duration(time.Millisecond*80))
    defer cancel()

    req, err := http.NewRequestWithContext(ctx, http.MethodGet,
"http://example.com", nil)

    // ...
```

- Zapoznaj się z dokumentacją pod adresem: https://pkg.go.dev/context oraz przykładami: https://pkg.go.dev/context#pkg-examples użycia pakietu `context`. Jest to jeden z najczęściej wykorzystywanych pakietów języka Go.

Pakiet crypto

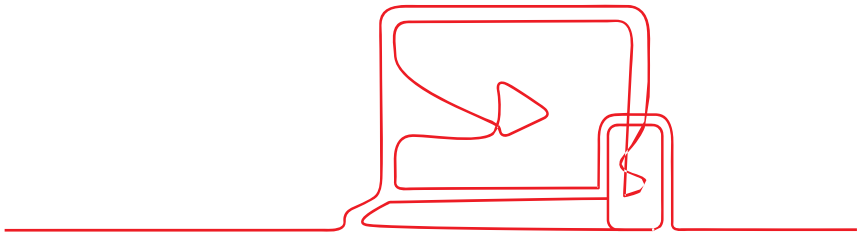
- Pakiet realizuje wiele algorytmów szyfrowania i funkcji skrótu, np.
  - aes (AES encryption)
  - ecdsa (Elliptic Curve Digital Signature Algorithm)
  - hmac (Keyed-Hash Message Authentication Code)
  - md5 (MD5 hash algorithm)
  - rand (cryptographically secure random number generator)
  - rsa (RSA encryption)
  - sha256 (SHA224 and SHA256 hash algorithms)

```go
import "crypto/sha256"

h := sha256.New()
h.Write([]byte("hello world\n"))
fmt.Printf("%x", h.Sum(nil))
// Output
// a948904f2f0f479b8f8197694b30184b0d2ed1c1cd2a1ec0fb85d299a192a447
```

- Zapoznaj się z dokumentacją pakietu `crypto` pod adresem: https://pkg.go.dev/crypto

Pakiet encoding

- Pakiet implementuje wiele formatów zapisu, np:
  - Base64
  - Binary
  - Csv
  - Hex
  - Json
  - xml
- Jednym z najczęstszych zastosowań jest parsowanie do/z formatu JSON.



Kodowanie JSON

- Aby otrzymać zakodowany zgodnie z formatem JSON ciąg bajtów, należy zapisać wymaganą strukturę w postaci typu map lub `struct`

```go
func main() {
    params := map[string]any{
        "key1": "text",
        "key2": 10,
        "key3": true,
    }

    j, err := json.Marshal(params)
    if err != nil {
        panic(err)
    }

    fmt.Println(string(j))
    // Output
    // {"key1":"text","key2":10,"key3":true}
}
```

Kodowanie JSON

- Taki sam rezultat można uzyskać wykorzystując typ `struct`

```
type Params struct {
    Key1 string
    Key2 int
    Key3 bool
}

func main() {
    params := Params{}

    j, err := json.Marshal(params)
    if err != nil {
        panic(err)
    }

    fmt.Println(string(j))
    // Output
    // {"key1":"text","key2":10,"key3":true}
}
```

- Dopuszczalne są także inne konstrukcje, np.

```
    k := map[string]any {
        "key1": "string1",
    }
    params := []any{k, "value"}

    j, err := json.Marshal(params)
    // ...
    // Output
    // [{"key1":"string1"},"value"]
```

Dekodowanie JSON

- Dekodowanie ciągu bajtów w formacie JSON jest również możliwe w połączeniu z typem `map` lub `struct`

```
jsonString := `{"key1":"text","key2":10,"key3":true}`

params := make(map[string]any)
if err := json.Unmarshal([]byte(jsonString), &params); err != nil {
    panic(err)
}

fmt.Println(params["key1"])
// Output
// text
```

Dekodowanie danych JSON z io.Reader

- Jeśli źródłem danych do zdekodowania jest `io.Reader`, np. podczas pobierania zawartości pakietu HTTP, należy użyć metody `json.NewDecoder()`

```
data := []byte(`{"key1":"text","key2":10,"key3":true}`)

// Utworzenie bufora spełniającego io.Reader
r := bytes.NewReader(data)

params := make(map[string]any)
if err := json.NewDecoder(r).Decode(&params); err != nil {
    panic(err)
}

fmt.Println(params["key1"])
// Output
// text
```

Kodowanie danych JSON do io.Writer

- Jeśli wynikiem zakodowania danych ma być `io.Writer`, użyj poniższej konstrukcji.

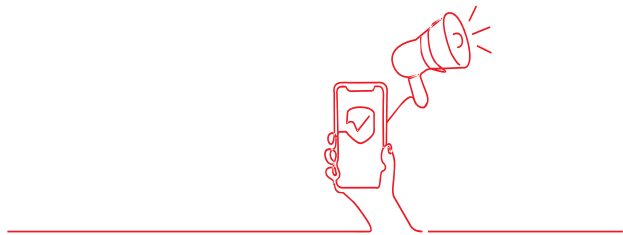
```
    params := map[string]any{
        "key1": "text",
        "key2": 10,
        "key3": true,
    }

    b := &bytes.Buffer{}
    if err := json.NewEncoder(b).Encode(&params); err != nil {
        panic(err)
    }

    fmt.Println(b.String())
    // Output
    // {"key1":"text","key2":10,"key3":true}
```

- Zapoznaj się z dokumentacją https://pkg.go.dev/encoding oraz przykładami https://pkg.go.dev/encoding/json#pkg-examples pakietu `encoding` oraz `encoding/json`

Pakiet flag

- Pakiet flag umożliwia parsowanie flag command-line, przekazywanych do programu w parametrze wywołania
- Flagi mogą być różnych typów, np. `string`, `bool`, `int`
- Dozwolone są konstrukcje:

```
-flag
--flag
-flag=x
-flag x  // nie dotyczy typu 'bool'
```



Pakiet flag

```
var (
        nFlag = flag.Int("n", 10, "max concurrent connections")
        sFlag = flag.String("url", "", "URL address")
    )

    flag.Parse()
    fmt.Println(*nFlag, *sFlag)
$ ./example --help
Usage of ./example:
  -n int
        max concurrent connections (default 10)
  -url string
        URL address
$ ./example -n 20 -url example.com
20 example.com
```

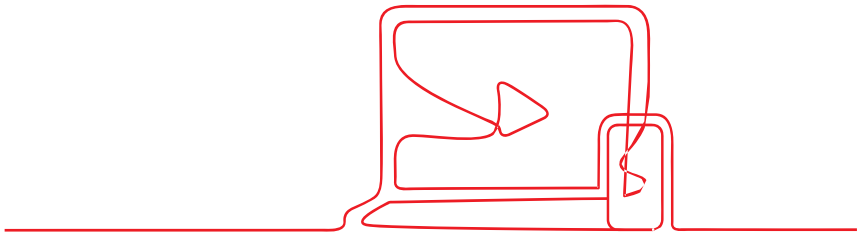
- Zapoznaj się z dokumentacją https://pkg.go.dev/flag oraz przykładami https://pkg.go.dev/flag#pkg-examples

Pakiet fmt

- Pakiet `fmt` służy do formatowania tekstu, a nie do logowania zdarzeń działającej aplikacji (do tego służy `log` opisany niżej)
- Implementuje funkcje wejścia-wyjścia znane z innych języków programowania (np. `printf`)
- Możesz użyć m.in. takich parametrów jak
  - `%s` - ciąg znaków typu *string*
  - `%d` - *integer* o podstawie 10 (decimal)
  - `%h` - *integer* o podstawie 16 (hex)
  - `%f` - *float*
  - `%t` - *bool*
- Zapoznaj się z pełną dokumentacją https://pkg.go.dev/fmt

Pakiet fmt

- Wybrane funkcje pakietu
  - Printf(format string, a …any) (n int, err error)
  - Println(a …any) (n int, err error)
  - Sprintf(format string, a …any) string
  - Fscanln(r io.Reader, a …any) (n int, err error)



Pakiet fmt

- Funkcja `Errorf(format string, a ...any) error` zwraca *error* zbudowany zgodnie z przekazanymi parametrami formatowania
- Możesz użyć tej funkcji i parametru %w do *owijania* (wrap) błędów występujących na wielu poziomach aplikacji. Więcej na ten temat dowiesz się w części omawiającej interfejs *error*

```
func RunTasks() error {
    // ...
    return fmt.Errorf("task limit exceeded (%d > %d)", tasks, taskLimit)
}

func InitWorker() error {
    // ...
    if err := RunTasks(); err != nil {
        // Jeśli 'err' przekazany przez parametr '%w' spełni interfejs 'error', to
        // zwrócony przez fmt.Errorf() typ implementuje metodę 'Unwrap()'
        return fmt.Errorf("worker handler error: %w", err)
    }
}

func main() {
    fmt.Println(InitWorker())
    // Output
    // worker handler error: task limit exceeded (10 > 9)
}
```

Pakiet log

- Pakiet `log` implementuje podstawowy logger umożliwiający zwracanie informacji o działaniu aplikacji
- Jest zaprojektowany aby zapewnić stabilną pracę pod dużym obciążeniem i z wielu gorutyn jednocześnie
- Do komunikatów mogą być dołączone informacje o dacie, czasie, nazwie pliku źródłowego itp.
- Ważniejsze funkcje pakietu to
  - Fatal(v ...any)
  - Panic(v ...any)
  - Printf(format string, v ...any)
- Aby użyć loggera możesz natychmiast skorzystać z funkcji Printf

```
log.Printf("INFO: connection accepted.")
// Output
// 2023/01/01 23:00:00 INFO: connection accepted.
```

Pakiet log

- Aby zdefiniować formatowanie należy użyć typu `log.Logger*

```
var (
    // log.Lshortfile oznacza dołączenie nazwy pliku oraz numeru linii
    logger = log.New(os.Stdout, "logger: ", log.Lshortfile)
)

logger.Print("Hello, log file!")
// Output
// logger: example_test.go:13: Hello, log file!
```

- Typ `log.Logger` można przekazać przez `context` zgodnie z przykładem omówionym w rozdziale Pakiet context

Pakiet log

- Podstawowy pakiet `log` nie umożliwia definiowania poziomów komunikatów, np. INFO, ERROR itp.
- Nie umożliwia także formatowania wyjścia np. w formacie JSON
- Od niedawna dostępny jest *eksperymentalny* pakiet `slog`, który docelowo powinien zastąpić `log` w bibliotece standardowej
- Aktualnie pakiet ten jest dostępy w niezależnym od biblioteki standardowej repozytorium: `golang.org/x/exp/slog`
- Pakiety w tym repozytorium mogą ulec zmianie lub zostać całkowicie wycofane
- Zapoznaj się z dokumentacją `slog` oraz `log` pod adresami: https://pkg.go.dev/golang.org/x/exp/slog, https://pkg.go.dev/log

Pakiet net

- Pakiet `net` to bardzo złożony pakiet implementujący wiele funkcji i typów do niskopoziomowego zarządzania połączeniami sieciowymi
- Kilka przykładowych typów
  - Addr - reprezentuje adres w internecie
  - Conn - interfejs połączenia sieciowego, zgodny z *io.Reader* oraz *io.Writer*, używający adresacji zgodnych z typem Addr
  - Dialer - wykonuje połączenie pod wskazany adres w sieci
  - Listener - interfejs nasłuchującej strony połączenia sieciowego
- Dodatkowo dostępnych jest kilka pakietów w obrębie pakietu net
  - net/http
  - net/mail
  - net/smtp
  - net/url
- Zapoznaj się z dokumentacją pakietu `net` https://pkg.go.dev/net#pkg-overview oraz `net/http` https://pkg.go.dev/net/http

Pakiet net/http

- Pakiet `net/http` implementuje typy i funkcje niezbędne do obsługi klienta oraz serwera protokołu HTTP
- Przykład prostego zapytania GET

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := io.ReadAll(resp.Body)
// ...
```

Pakiet net/http

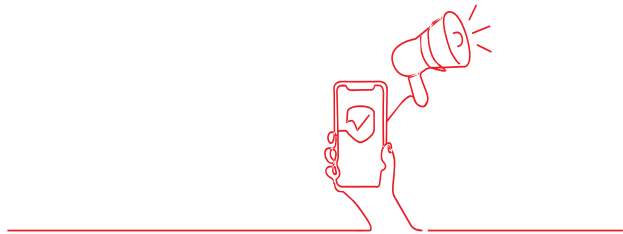
- Aby sterować wszystkimi parametrami requestu HTTP należy zainicjować strukturę http.Client oraz skorzystać z `http.NewRequest`

```go
// W strukturze http.Client można zdefiniować dodatkowe parametry, np. Timeout
client := &http.Client{}

// Parametr 'body' ustawiony na 'nil', ponieważ nie przesyłamy żadnych danych
req, err := http.NewRequest(http.MethodPost, URL, nil)
if err != nil {
    return err
}

// W tym miejscu zapytanie nie zostało jeszcze wysłane. Nadal można wpływać na zawartość
// requestu, np. dodając nagłówki itp.

// Wysłanie zapytania
resp, err := client.Do(req)
if err != nil {
    return err
}
```

- Zapoznaj się z przykładami w repozytorium tej dokumentacji
- https://github.com/grupawp/pjatk-akademia-programowania/blob/main/prezentacja/przyklady/stdlib/net.http.01/example.go
- https://github.com/grupawp/pjatk-akademia-programowania/blob/main/prezentacja/przyklady/stdlib/net.http.02/example.go

Pakiet os

- Pakiet os udostępnia abstrakcyjną warstwę dostępu do funkcji systemu operacyjnego
- Przykładowe funkcje to:
  - Chdir(dir string) error
  - Mkdir(name string, perm FileMode) error
  - Rename(oldpath, newpath string) error
  - Create(name string) (*File, error)
  - Open(name string) (*File, error)



Pakiet os

- W pakiecie os dostępne są m.in. typy
  - os.File - operacje na pliku, np. Chmod, Read, Seek
  - os.Process - operacja na procesach, np. Kill, Wait, Signal
- Zapoznaj się z dokumentacją pakietu os https://pkg.go.dev/os
- Przykłady zastosowania pakietu https://pkg.go.dev/os#pkg-examples

Pakiet strconv - konwersja z tekstu

- `strconv` zawiera funkcje umożliwiające konwersję typów prostych z i do postaci tekstowej
- Konwersji z postaci łańcucha służy rodzina funkcji `Parse*`:

```
    // func ParseBool(str string) (bool, error)
    b, err := strconv.ParseBool("True")
    // func ParseInt(s string, base int, bitSize int) (i int64, err error)
    i, err := strconv.ParseInt("1337", 10, 0)
    // func ParseFloat(s string, bitSize int) (float64, error)
    f, err := strconv.ParseFloat("1337", 64)
```

- W przypadku liczb całkowitych *base* jest podstawą, z jaką zapisana jest liczba
  - gdy ta ustawiona zostanie na zero, łańcuch może poprzedzać prefix (*“0b”*, *“0o”*, *“0x”*)
- W przypadku liczb całkowitych `bitSize` to “szerokość” docelowego typu
  - zawsze zwrócony będzie `int64`, ale zyskujemy gwarancję możliwości konwersji do węższego typu
  - w przypadku zera będzie to zwykły `int`, ale np. 8 da `int8`, 16 - `int16`, etc.
- W przypadku liczb zmiennoprzecinkowych, zawsze dostaniemy `float64`
  - `bitSize` ustawiony na 32 zapewni jednak, że po konwersji do float32 wartość nie zmieni się

Pakiet strconv - konwersja do tekstu

- Konwersji z typów prostych do tekstu dokonamy za pomocą funkcji rodziny `Format*`:

```
// func FormatBool(b bool) string
s := strconv.FormatBool(true)
// func FormatInt(i int64, base int) string
s := strconv.FormatInt(1337, 16)
// func FormatFloat(f float64, fmt byte, prec, bitSize int) string
s := strconv.FormatFloat(1337, 'f', -1, 64)
```

- W przypadku liczb całkowitych możemy podać podstawę w zakresie 2 do 36 (cyfry, litery a do z)
- Dla liczb zmiennoprzecinowych możemy określić format, np.:

```
'e' -> 1.337e+03, 'E' -> 1.337E+03, 'f' -> 1337
'g' -> mieszane 'e' z 'f', zależnie od wykładnika
'G' -> mieszane 'E' z 'f', zależnie od wykładnika
```

- Precyzja (`prec`) określa zwykle liczbę cyfr - tutaj *-1* oznacza *jak najmniej, by oddać liczbę*
- `bitSize` wskazuje na szerokość pierwotnej liczby, wpływa na zaokrąglenie

Pakiet strconv - ułatwienia, Quote/Unquote

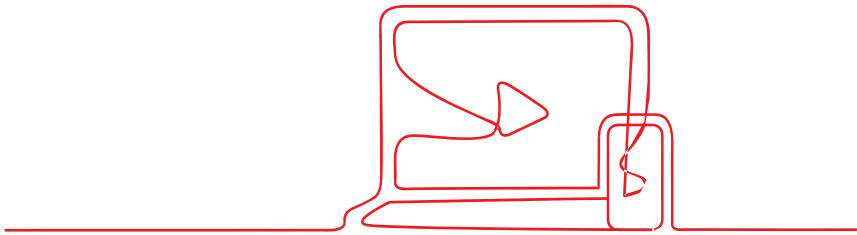
- Możemy też skorzystać z prostszych, znanych z innych języków, funkcji `Atoi` oraz `Itoa`:

```
i, err := strconv.Atoi("1337")
s := strconv.Itoa(1337)
```

- Czasem też prościej i szybciej skorzystać z `fmt.Sprintf`...
- `strconv.Quote` zwraca łańcuch w postaci odpowiadającej literałowi łańcuchowemu, razem ze znakami cudzysłowów:

```
strconv.Quote(`"Line one\nLine two"`) -> "\"Line one\\nLine two\""
```

- Odwrotną operacją jest `strconv.Unquote`



Pakiet strings

- Pakiet `strings` daje możliwość wykonywania operacji na łańcuchach.
- Sporo funkcji pokrywa się z tymi z pakietu `bytes`:
  - Compare(a, b string) int
  - Contains(b, substr string) bool
  - HasPrefix(s, prefix string) bool
  - Index(s, substr) int
  - ReplaceAll(s, old, new string) string
  - Split(s, sep string) []string
  - ToLower(s string) string

Pakiet strings - Reader

- Jak w `bytes`, także i tutaj znajdzie się `io.Reader` (i krewni) - konkretnie `strings.Reader`:

```
r := strings.NewReader("tylko czyste C")
c, err := io.ReadAll(r) // []byte
```

- Pusty `strings.Reader` zachowuje się jak Reader stworzony z pustego łańcucha

```
var s strings.Reader
r := strings.NewReader("")
```

Pakiet strings - Builder

- Go ma też swój string builder - `strings.Builder`!
  - Wystarczy zdefiniować pusty Builder...
  - pisać do niego przy użyciu `Write` (tak, to `io.Writer`)/`WriteString`/`...Byte`/`...Rune`
  - ... i pobrać gotowy łańcuch - wywołaniem `String()`

```
var b strings.Builder

for i := 5; i > 0; i-- {
    fmt.Fprintf(&b, "zostało %d... ", i)
}
b.WriteString("zakąska")

fmt.Println(b.String())
```

Pakiet sync - Mutex

- Zdarza się, że dostęp do zasobów musi być synchronizowany - środków do tego dostarcza pakiet `sync`
- Gdy nie chcemy, by dany fragment kodu mógł wykonywać się równolegle, możemy wykorzystać muteks – `sync.Mutex`:

```
mu sync.Mutex
...
mu.Lock()
wypłaćStówkę()
mu.Unlock()
```

[Go Playground - uwaga!]

Pakiet sync - RWMutex

- W przypadku, gdy mamy przewagę równoległych odczytów, można skorzystać z `sync.RWMutex`
- W danej chwili albo jedna gorutyna pisze, albo wiele – czyta
- "Writer" korzysta z `Lock/Unlock`, odczyty "owinięte" są zaś przez `RLock/RUnlock`:

```
mu sync.RWMutex
...
func ConcurrentReader() {
    mu.RLock()
    defer mu.RUnlock()
}

func Writer() {
    mu.Lock()
    defer mu.Unlock
}
```

- Uwaga: RWMutex brzydko skaluje się przy dużej liczbie procesorów

Pakiet sync - Map

- Wbudowane mapy są świetne, nie pozwalają jednak na równoczesny odczyt i zapis
- Gdy *runtime* wykryje taki przypadek, program zostanie zatrzymany z komunikatem: *fatal error: concurrent map read and map write*
- Rozwiązaniem może być ręczna synchronizacja dostępu do mapy, lub… użycie `sync.Map`

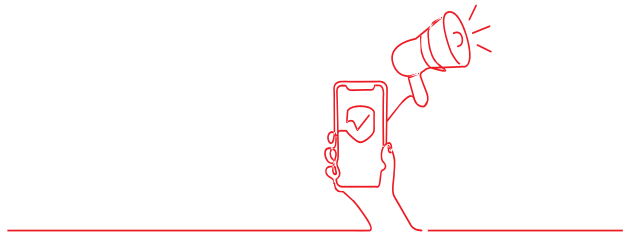
```
    var m sync.Map

    m.Store("dwa", "kopytka")
    var s string
    s, ok := m.Load("dwa")
    // Build failed!
    // cannot use m.Load("dwa") (value of type any) as string value in assignment:
need type assertion
```

Pakiet sync - Map

- Co się stało? `sync.Map` wyrzuca kontrolę typów przez okno!

```
sync.Map ~ map[any]any

val, ok := m.Load("dwa")
s := val.(string)
```

- To specyficzny typ danych, polecany w dwóch przypadkach:
  1. gdy dana wartość zapisywana jest raz, a potem tylko czytana (np. cache)
  2. gdy różne gorutyny operują na rozłącznych zestawach kluczy



Pakiet sync - WaitGroup

By poczekać na wyniki uruchomionych gorutyn, możemy użyć kolejnej konstrukcji z pakietu - `sync.WaitGroup`

- Deklarujemy, na ile gorutyn czekamy - `Add`, po czym "przysypiamy" – `Wait`
- Z kolei każda z uruchomionych gorutyn na koniec przetwarzania wywołuje metodę `Done`

```
func robotnik(wg *sync.WaitGroup, i int) {
    fmt.Println("oho, robótka", i)
    wg.Done()
}

...
ilePrac := 12
wg.Add(ilePrac)
for i := 1; i <= ilePrac; i++ {
    go robotnik(&wg, i)
}
wg.Wait()
fmt.Println("zrobione")
```

- Jaki byłby wynik, gdyby usunąć kod dotyczący `sync.WaitGroup`?

Go Playground

Pakiet sync - uwagi

- Warto zaznaczyć, że Go faworyzuje komunikację z użyciem kanałów
- *Don't communicate by sharing memory, share memory by communicating.*
- W przypadku większości typów pakietu `sync` zmiennych nie wolno kopiować po pierwszym użyciu!
- Często spotykana konstrukcja `defer Unlock()` zapewnia bezpieczeństwo (gwarancja odblokowania), jednak czas "trzymania" obiektu może być wydłużony

Pakiet time

- Pakiet `time` zawiera zestaw narzędzi dotyczących czasu - nie tylko odmierzania, ale też drukowania, parsowania, etc.
- Podstawową operacją jest pobranie aktualnego czasu:

```
t := time.Now()
fmt.Println(t)
// Output
// 2019-11-01 13:00:00 +0000 UTC
```

- Czas zwracany jest w postaci struktury typu `time.Time` - ta zaś oferuje bogaty wachlarz metod:

```
fmt.Println(t.Year(), t.Month(), t.Day(), t.Weekday())
// Output
// 2019 November 1 Friday
```

Pakiet time - Unix timestamp, Equal

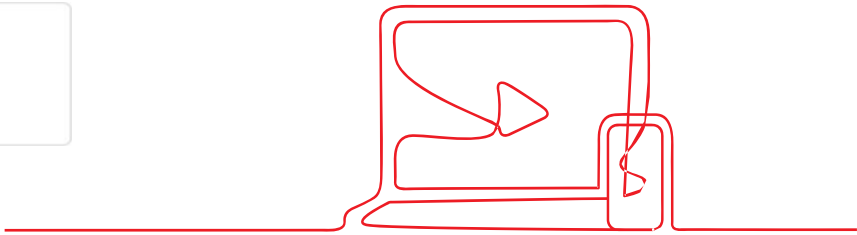
- Często pojawia się potrzeba przetwarzania czasu w postaci Unix timestamp:

```
fmt.Println(t.Unix(), t.UnixNano())
// Output
// 1572613800 1572613800000000000
fmt.Println(time.Unix(1572613800, 0))
// Output
// 2019-11-01 13:00:00 +0000 UTC
```

- Struktura time.Time zawiera więcej elementów, stąd do porównywania czasu najlepiej używać metody `Equal`

```
t.Equal(wczoraj)
```

```
// Output
// "niestety - nie"
```



Pakiet time - Duration

- Poza “punktem w czasie”, pakiet time oferuje też typ opisujący okres - `time.Duratio`

```
var zostało time.Duration = 2 * time.Hour
kwadrans := 15 * time.Minue
```

- Czas można odejmować (Sub) oraz dodawać (Add)

```
start := time.Now()
kopMonetę()
koniec := time.Now()
...
fmt.Println("Kopanie trwało %v\n", koniec.Sub(start))
```

- Można też sprawdzić, ile czasu upłynęło od pewnego momentu (lub ile czasu zostało)

```
start := time.Now()
koniec := time.Since(start)

zostało := time.Until(koniecŚwiata) // ups, do 290 lat!
```

Pakiet time - formatowanie i parsowanie

- Pakiet `time` dostarcza też narzędzi do formatowania czasu:

```
t.Format("2006-01-02T15:04:05")
// 2019-11-01T13:10:00
t.Format("01-02-2006 15:04")
// 11-01-2019 13:10
t.Format(time.UnixDate) // "Mon Jan _2 15:04:05 MST 2006"
// Fri Nov  1 13:10:00 UTC 2019
t.Format(time.RFC822Z)  // "02 Jan 06 15:04 -0700"
// 01 Nov 19 13:10 +0000
```

- Wybrane w wywołaniu wzorce nie są przypadkowe - muszą odnosić się do sztywno ustalonej daty:

```
Mon Jan 2 15:04:05 2006 MST
  0   1 2  3  4  5    6  -7
```

- Ten sam wzorzec używany jest przy parsowaniu:

```
layout := "2006-Jan-02"
t, err := time.Parse(layout, "2023-Jan-01")
```

Pakiet time - Timer, Ticker

- Możemy również tworzyć timery (wybudzane raz):

```
timer := time.NewTimer(1 * time.Second)
go func() {
    <-timer.C
    fmt.Println("Minęła sekunda")
}()
// Output
// Minęła sekunda
```

- Oraz tickery - budzone co określony czas:

```
ticker := time.NewTicker(1 * time.Second)
go func() {
    for {
        <-ticker.C
        fmt.Println("Minęła sekunda")
    }
}()
// Output
// Minęła sekunda
// Minęła sekunda
// ...
```

Go Playground