



POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

PAKIETY, EKSPORTY, GO
MOD



Go modules



- Moduł jest to kolekcja powiązanych ze sobą pakietów w repozytorium kodu źródłowego
- Definicja modułu oraz pakietów znajduje się w pliku go.mod





Tworzenie nowego modułu

Tworzenie nowego modułu



→ Tworzymy folder dla naszego projektu oraz przechodzimy do jego lokalizacji

```
$ mkdir pjatk_project  
$ cd pjatk_project
```

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

```
$ go mod init pjatk_project
```

Tworzenie nowego modułu



→ W wyniku metody init w folderze projektu został wygenerowany plik go.mod

```
pjatk_project/  
└─ go.mod
```

```
module pjatk_project  
  
go 1.19
```

→ Plik go.mod zawiera aktualnie informacje

- Nazwa modułu
- Wersja GO



Dodawanie paczek do projektu

Dodawanie paczek do projektu



- Program który napiszemy będzie miał za zadanie generować oraz wyświetlać unikalny identyfikator
- W tym celu wykorzystamy bibliotekę UUID dostarczoną przez Google

<https://github.com/google/uuid>

Dodawanie paczek do projektu



- Instalujemy zależność z pomocą narzędzia `go get`

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

```
module pjak_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 paczek do projektu

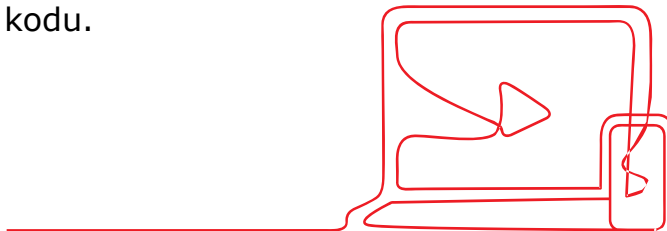


→ W projekcie pojawił się plik `go.sum`

```
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 listuje informacje o checksumach zależności projektu, by mieć pewność że niezależnie od czasu i środowiska będziemy korzystali z tego samego kodu.



Dodawanie paczek do projektu



→ Wykorzystajmy paczkę w programie

```
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



→ Instalacja nowej paczki

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

→ Zmiany w kodzie

```
package main

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

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

Zmiany zależności



→ Aktualizacja pliku go.mod oraz go.sum

```
$ go mod tidy
```

→ W wyniku tej operacji nieużywana już paczka UUID została usunięta a nowa randstr dodana w plikach go.mod oraz go.sum:

```
module pjatk_project
```

```
go 1.19
```

```
require github.com/thanhpk/randstr v1.0.4
```

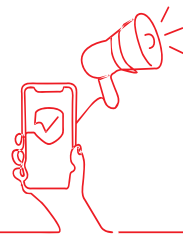
```
github.com/thanhpk/randstr v1.0.4 h1:IN78qu/bR+My+gHCvMEXhR/i5oriVHcTB/BJJIRTsNo=  
github.com/thanhpk/randstr v1.0.4/go.mod h1:M/  
H2P1eNLZz1DwAzpkkkUvoyNNMbZRGhESZuEQk3r0U=
```

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
```

Tworzenie własnych pakietów i eksportowanie nazw



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

```
package token

import "github.com/thanhpk/randstr"

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

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"
    "pjak_project/token"
)

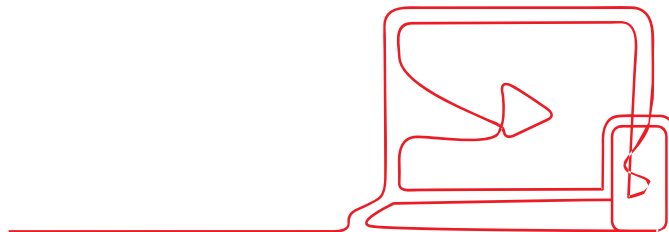
func main() {
    uniqueToken := token.generate(16)
    fmt.Printf("Twój unikalny token to: %s", uniqueToken)
}
```


Tworzenie własnych pakietów i eksportowanie nazw



→ Uruchamiamy projekt

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



Tworzenie własnych pakietów i eksportowanie nazw

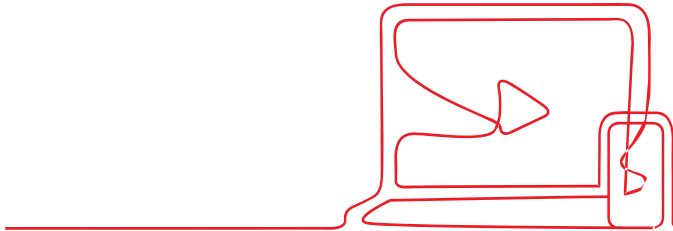


- W konsoli pojawił się błąd, ponieważ nasza funkcja `generate` nie została eksportowana i nie jest widoczna nigdzie poza pakietem `token`.
- Aby eksportować należy zmienić nazwę funkcji w `generator.go` tak by zaczynała się z wielkiej litery

```
func Generate(size int) string
```

- Edytujemy wywołanie funkcji w pliku `main.go`

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



Tworzenie własnych pakietów i eksportowanie nazw



Podsumowanie

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



POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

METODY, INTEFEJSY





Metody

Multiple results named return value



- Możliwość przypisania kilku wartości jednocześnie (prawie)
- Staramy się by **error był ostatnim typem**, jeśli logika tego nie wymaga zwracamy zerowe wartości

```
func main() {  
    x, y, z := multipleValues(2137)  
    fmt.Println(x, y, z)  
    fmt.Println(countVal(420, []int{42, 420, 1337, 666, 0, -1, 420}))  
}  
  
func multipleValues(v int) (int, float64, string) {  
    f := float64(v)  
    s := strconv.Itoa(v)  
    return v, f, s  
}  
  
func countVal(i int, values []int) (firstIndex int, count int) {  
    firstIndex = -1  
    for idx, val := range values {  
        if val != i {  
            continue  
        }  
        if firstIndex < 0 {  
            firstIndex = idx  
        }  
        count++  
    }  
    return // same as: return firstIndex, count  
}
```

Metody



→ Możliwość przypisania kilku wartości jednocześnie (prawie)

```
type MyStruct struct {
    privateField string
}

func (m MyStruct) myMethod() {
    fmt.Println(m)
    m.privateField = "42" // ineffective assignment to field
}

func (m *MyStruct) myNewMethod() {
    fmt.Println(m)
    m.privateField = "42"
}

func main() {
    var m MyStruct
    m.myMethod()
    m.myNewMethod()

    var mp *MyStruct
    mp.myMethod() // Panic!
    mp.myNewMethod()
}

// Output:
// {}
// &{}
// panic: runtime error: invalid memory address or nil pointer dereference
```

New



→ New zwraca wskaźnik na podany typ

```
func main() {  
    var m MyStruct  
    m.myMethod()  
    m.myNewMethod()  
  
    mp :=new(MyStruct)  
    mp.myMethod() // YAY :D!  
    mp.myNewMethod()  
}
```

```
// Output:  
// {}  
// &{}  
// {}  
// &{}  

```


Make



→ Alokuje pamięć pod dany typ: slice, map lub chan

```
func main() {  
    x := make([]int, 2)  
    y := make([]int, 2, 3)  
    z := make([]int, 0, 3)  
    fmt.Println(x, len(x), cap(x))  
    fmt.Println(y, len(y), cap(y))  
    fmt.Println(z, len(z), cap(z))  
}
```

```
// Output:  
// [0 0] 2 2  
// [0 0] 2 3  
// [] 0 3
```

Więcej można przeczytać tu: <https://pkg.go.dev/builtin>



INTERFEJSY

Interfejsy



- Typ
- Zbiór metod tworzących pewną sygnaturę
- Jeśli struktura posiada pełen zestaw metod danego interfejsu to może być użyta jako ten interfejs

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

Interfejsy



- Nie ma potrzeby jawnej deklaracji przypisania struktury do interfejsu
- Struktura może spełniać wiele interfejsów

```
// Struktura posiada metody Read i Write, więc automatycznie spełnia  
// interfejs ReadWriter  
type Connection struct {  
}  
  
func(c *Connection)Read(p []byte) (n int, err error) {  
    // ...  
}  
  
func(c *Connection)Write(p []byte) (n int, err error) {  
    // ...  
}
```

Interfejsy



```
type Data []byte

type Fetcher interface {
    Fetch() (Data, error)
}

type SSHClient struct {
    User, Pass string
}

func (c *SSHClient) Fetch() (Data, error) {
    // TODO: Implement the logic
    return nil, nil
}

type HTTPClient struct {
    Secure bool
    Host, Token string
}

func (c HTTPClient) Fetch() (Data, error) {
    // TODO: Implement the logic
    return nil, nil
}

func fakeService(f Fetcher) {
    f.Fetch()
}
```

```
func main() {
    http := HTTPClient{
        Secure: true,
        Host:    "pudelek.pl",
    }
    ssh := SSHClient{
        User: "root",
        Pass: "toor",
    }
    fakeService(http)
    fakeService(&ssh)
}
```

Empty interface any



- Szczególnym przypadkiem jest typ `interface{}` lub jego alias `any`
- Dowolny typ spełnia ten interfejs
- Puste interfejsy powinny być używane w szczególnych sytuacjach, gdyż nie zapewniają kontroli typu (należy to zrobić samodzielnie)

```
type Event any

type UserProfile struct {}
type Action struct {}

func Parse(e Event) {
    switch x := e.(type) {
    case UserProfile:
        // ...
    case Action:
        // ...
    default:
        fmt.Println("Not recognized")
    }
}
```

Dobre praktyki z interfejsami



- Interfejs spełnia jedną logikę związaną z jego nazwą
- Nazwa najczęściej jest rzeczownikiem, np. Reader, Writer, Stringer
- Przyjmuj interfejsy zwracaj struktury
- Projektując bibliotekę postaraj się by akceptowała interfejsy, by dać użytkownikom możliwość przekazywania własnych struktur oraz uprościć testowanie
- Pozwoli to precyzyjnie określić zakres odpowiedzialności
- Struktura może mieć szerszy zestaw metod niż interfejs który spełnia
- Interfejsy używane są w celu odizolowania warstw aplikacji, np. jako mock
- Za pomocą interfejsów tworzymy też typy generyczne... ale o nich później 🙄

Dobre praktyki z interfejsami



```
// Struktura zdefiniowana w zewnętrznym pakiecie
type Profil struct{
func (p *Profil) SetName() {...}
func (p *Profil) SetAge() {...}

// Interfejs zdefiniowany po stronie "klienta" struktury.
// Załóżmy, że logika nie wymaga korzystania z metody SetAge()
// Aby zmniejszyć zakres odpowiedzialności tworzymy interfejs
// zawierający tylko niezbędną funkcję SetName()
type Namer interface {
    SetName()
}

func ChangeName(n Namer) {
    n.SetName()
}

func main() {
    user := Profil{}
    ChangeName(user)
}
```


Error



- Typ error **nie jest stringiem, jest interfejsem!** Dzięki temu możemy wstrzyknąć dowolny typ spełniający ten interfejs, a następnie po asercji typu obsłużyć bardziej złożoną logikę z nim związaną.

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

Przykład



```
var ErrSentinel = errors.New("something went wrong 🙄")

type customError struct {
    elapsed time.Duration
}

func (c *customError) Error() string {
    return fmt.Sprintf("timed out after %v", c.elapsed)
}

func someFunc(custom bool) error {
    if !custom {
        return ErrSentinel
    }
    return &customError{
        elapsed: time.Second * 42,
    }
}

func myService(custom bool) error {
    if err := someFunc(custom); err != nil {
        return fmt.Errorf("myService returned err: %w", err)
    }
    return nil
}

// Output
//true myService returned err: timed out after 42s
//false myService returned err: something went wrong 🙄
```

Więcej o errorach



→ <https://pkg.go.dev/errors>

→ <https://go.dev/blog/error-handling-and-go>

→ <https://lukas.zapletalovi.com/posts/2022/wrapping-multiple-errors>

io.Reader



- Przykładem często wykorzystywanego interfejsu jest `io.Reader`
- Interfejs ten zawiera sygnaturę funkcji `Read(p []byte) (n int, err error)`

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

Przykład dostępny pod adresem:

<https://go.dev/play/p/nsrLn200zfG>



POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

Podstawy GO



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

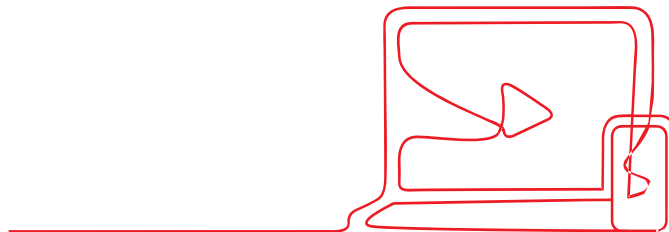
```
func x() (ret int) {  
    for i := 0; i < 5; i++ {  
        defer func() { ret++; fmt.Println(ret) }()  
    }  
    return 5  
}
```

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.

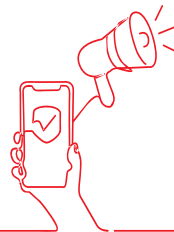


- Jeśli wartość interfejsu nie jest typu T, program zakończy się z błędem
- By tego uniknąć, możemy skorzystać z formy zwracającej dwie wartości
- Pierwsza to - jak poprzednio - zmienna konkretnego typu, druga - boolean - wskazuje na możliwość wykonania operacji boolean

```
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:  
        ...  
}
```

- 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

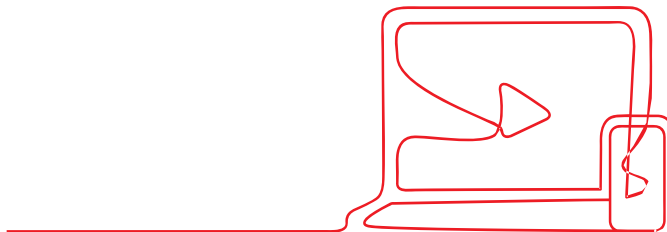


- Nie wszystkie funkcje muszą mieć nazwę - możemy też definiować równie użyteczne funkcje anonimowe

```
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



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

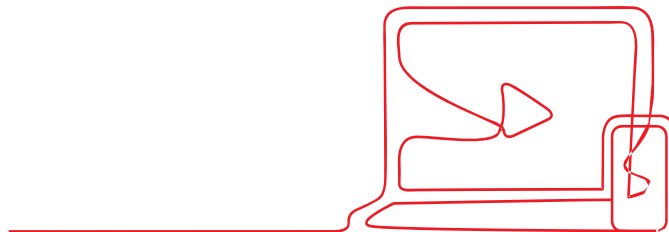


- Pakiet `bytes` umożliwia operacje na wycinku (slice) bajtów `[]byte`. Funkcje tego pakietu są analogiczne do funkcji pakietu `strings` który zostanie omówiony później.
- Przykładowe funkcje pakietu `bytes` to:
 - `Compare(a, b []byte) int`
 - `Contains(b, subslice []byte) bool`
 - `HasPrefix(s, prefix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

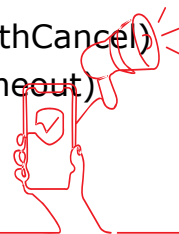
Pakiet context



- 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.

```
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



- ➔ Często 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`

```
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)

```
import "crypto/sha256"
```

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

```
// Output
```

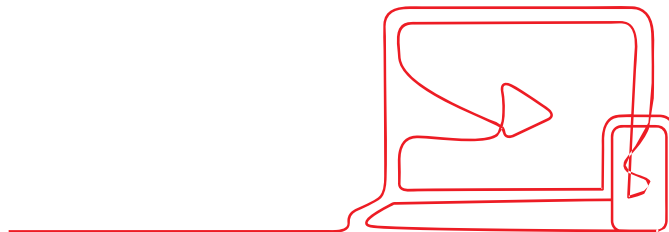
```
// a948904f2f0f479b8f8197694b30184b0d2ed1c1cd2a1ec0fb85d299a192a447
```

- 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

```
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 `json:"key_1,omitempty"`  
    Key2 int    `json:"key_2,omitempty"`  
    Key3 bool   `json:"key_3,omitempty"`  
}  
  
func main() {  
    params := Params{  
  
        j, err := json.Marshal(params)  
        if err != nil {  
            panic(err)  
        }  
  
        fmt.Println(string(j))  
        // Output  
        // {"key_1":"text","key_2":10,"key_3":true}  
    }
```

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}
```

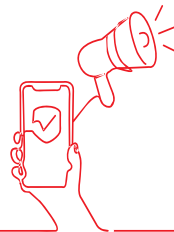
→ Zapoznaj się z dokumentacją <https://pkg.go.dev/encoding> oraz przykładami <https://pkg.go.dev/encoding/json#pkg-examples> pakietu encoding oraz encoding/json

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*
- `%v` - domyślny format (w tym `%+v` lub `%#v`)

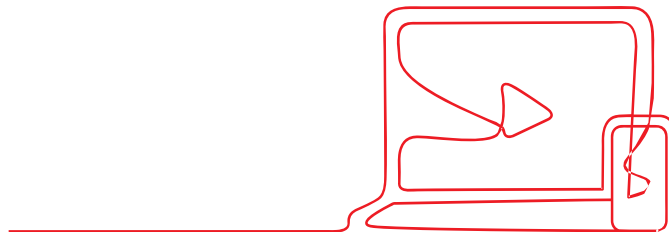
- Zapoznaj się z pełną dokumentacją <https://pkg.go.dev/fmt>

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 dostępny 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`

```
// 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" zwracanego 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 zmiennoprzecinkowych 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`:

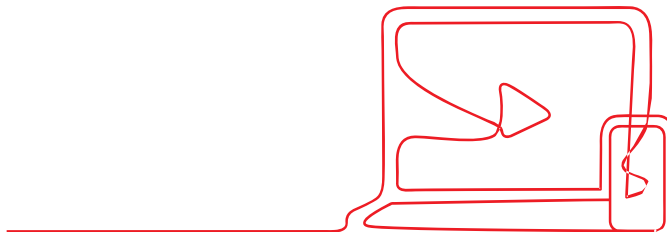
```
i, err := strconv.Atoi("1337")  
s := strconv.Itoa(1337)
```

→ 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()
```

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

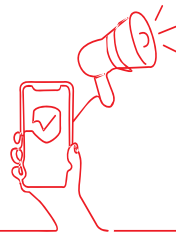


→ Co się stało? `sync.Map` wyrzuca kontrolę typów przez okno!

```
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`?

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

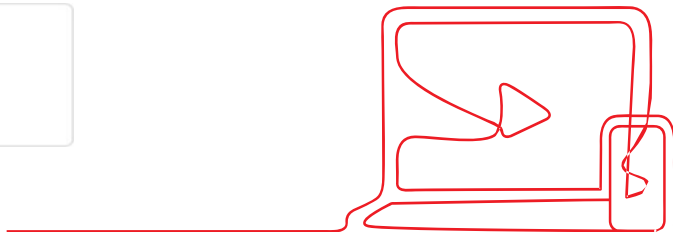


→ Często pojawia się potrzeba przetwarzania czasu w postaci Unix timestamp:

```
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.Duration`

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

→ 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 wzorec 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
// ...
```

Do NOT panic!



→ Panic powinien być używany tylko wtedy gdy to programista popełnił błąd, lub błędu nie da się sensownie obsłużyć

```
defer func() {  
    if r:=recover();r!=nil{  
        fmt.Println("Recovered:",r)  
    }  
}()  
panic("something is no yes")  
// Output  
// Recovered panic: something is no yes
```

Zadanko dla chętnych do poćwiczenia



`https://github.com/grupawp/akademia-programowania/tree/main/Golang/zadania/reddit`