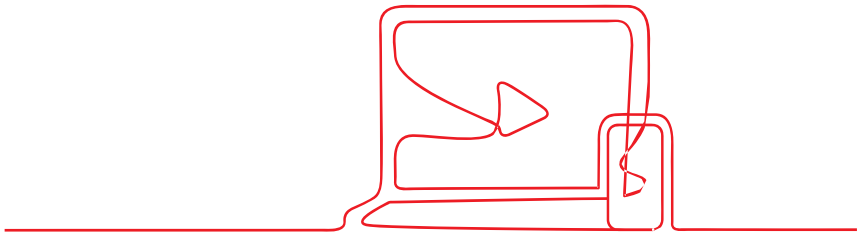
POLSKO-JAPOŃSKA
AKADEMIA TECHNIK
KOMPUTEROWYCH

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

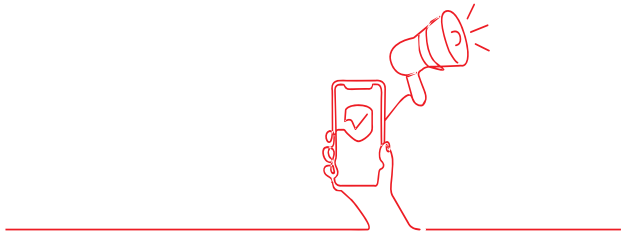
- W niektórych przypadkach kompilator może wywnioskować, jakiego typu zmienną definiujemy

- 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)
```

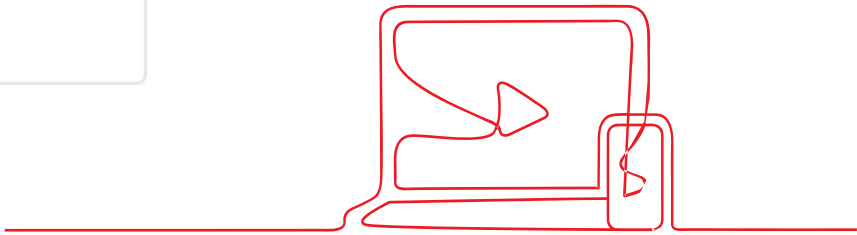
- Uzyskamy zmienną `s` typu string

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))
```

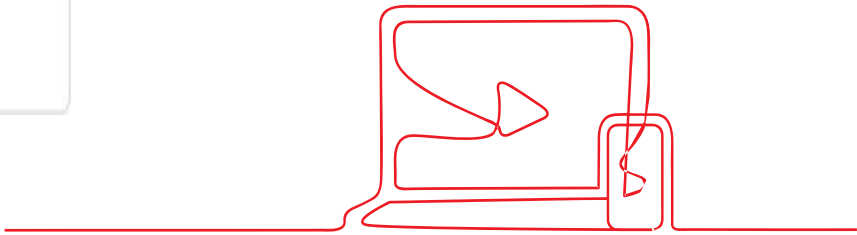
Funkcje anonimowe

- 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