Programski jezik Go (Golang)

1. Osnove

Program u Go-u sastoji se iz jednog ili više fajlova sa ekstenzijom .go organizovanih u pakete. U prvoj liniji svakog fajla neophodno je navesti naziv paketa kom on pripada. Svi .go fajlovi unutar jednog direktorijuma moraju pripadati istom paketu. Nakon toga sledi *import* sekcija u kojoj se navode paketi koji će biti korišćeni unutar trenutnog fajla. Import paketa koji se ne koristi prouzrokovaće grešku. Izvršavanje programa počinje unutar paketa *main* i njegove pripadajuće *main()* funkcije. U listingu 1 prikazan je jednostavan Go program

```
package main

import "fmt"

func main() {
   fmt.Println("Hello world")
}
```

Listing 1 - Hello world u Go-u

Ukoliko je potrebno izvršiti import više od jednog paketa preporučuje se upotreba stila prikazanog u listingu 2.

```
import (
   "fmt"
   "math"
)
```

Listing 2 - Import više paketa

Kako bi funkcija, promenljiva, konstanta itd. iz jednog paketa bila vidljiva unutar drugog paketa neophodno je da njen naziv počinje velikim slovom. Sva imena koja počinju malim slovom dostupna su samo unutar tog paketa.

1.1. Funkcije

Opšti oblik funkcije prikazan je u listingu 3. Nakon ključne reči *func* navodi se naziv funkcije. Funkcija može i ne mora imati parametre, a oni se navode tako što se prvo naznači naziv parametra, a zatim njegov tip.

```
func naziv_funkcije(param1 tip, param2 tip ...) (retval1 tip, retval2 tip ..) {
   //telo funkcije
}
```

Listing 3 - Opšti oblik funkcije

Ukoliko je više uzastopnih parametara istog tipa, on se može navesti samo iza poslednjeg takvog parametra u nizu (listing 4).

```
func add(x, y int) int {
   return x + y
}
```

Listing 4 - Parametri istog tipa

Funkcija može i ne mora imati povratnu vrednost, a može ih imati i više. Ukoliko je poslednje navedeno slučaj, tipovi povratnih vrednosti navode se unutar malih zagrada (listing 5).

```go
func swap(x, y int) (int, int) {
   return y, x
}
```

Listing 5 - Funkcija sa više od jedne povratne vrednosti

Povratne vrednosti funkcije mogu biti imenovane (listing 6) i u tom slučaju se tretiraju kao promenljive definisane na samom početku funkcije i ne navode se nakon *return* iskaza. Ovakav pristup preporučljiv je samo za kratke funkcije jer u suprotnom može dovesti do nečitljivosti koda.

```go
func add(x, y int) (sum int) {
   sum = x + y
   return
}
```

Listing 6 - Imenovana povratna vrednost funkcije

Poslednja povratna vrednost funkcije često je indikator greške. Go ne poseduje rad sa izuzecima i obrada grešaka vrši se uz pomoć interfejsa *error*. Funkcije koje mogu izazvati grešku bi trebale kao povratnu vrednost vratiti vrednost tipa *error* koja se nakon toga proverava. Ukoliko ta vrednost bude nil, znači da do greške nije došlo. Primer provere greške dat je u listingu 7.

```go
func main() {
   i, err := strconv.Atoi("42")
   if err != nil {
      fmt.Printf("couldn't convert number: %v\n", err)
      return
   }
   fmt.Println("Converted integer:", i)
}
```

Listing 7 - Rukovanje greškama

1.2. Promenljive i konstante

Deklaracija promenljivih vrši se navođenjem naziva jedne ili više promenljivih nakon ključne reči *var*. Promenljive se ujedno mogu i inicijalizovati. U tom slučaju nije neophodno navoditi tip promenljive, moguće je zaključiti ga iz vrednosti koja je dodeljena. Opisan način deklaracije i inicijalizacije promenljivih može se vršiti na nivou paketa ili na nivou funkcije. Skraćeni oblik inicijalizacije oblika *variable (type) := value* omogućen je samo na nivou funkcije. Listing 8 prikazuje rad sa varijablama.

```go
package main

import "fmt"

var a int
var b, c string = "hello", "world"
var d, e = 1, false

func main() {
   var i int
   var j = 5
   k := true
   //promenljive definisane na nivou paketa
   fmt.Println(a, b, c, d, e)
   //output: 0 hello world 1 false

   //promenljive definisane na nivou funkcije
   fmt.Println(i, j, k)
   //output: 0 5 true
   fmt.Printf("type of k is: %T", k)
   //output: type of k is: bool
}
```

Listing 8 - Rad sa varijablama

Rad sa konstantama isti je kao rad sa varijablama, uz zamenu ključne reči *var* rečju *const*. Skraćeni oblik inicijalizacije nije moguće primeniti na konstante. Kada je potrebno uzastopno deklarisati više promenljivih ili konstanti, deklaracije je moguće grupisati. Primer grupisanja deklaracije konstanti dat je u listingu 9.

```go
const (
   Pi = 3.14
   E = 2.72
)
```

Listing 9 - Grupisanje deklaracije konstanti

1.3. Prosti tipovi podataka

- bool
- string
- int, int8, int16, int32 (rune), int64
- uint, uint8 (byte), uint16, uint32, uint64, uintptr
- float32, float64
- complex64, complex128

Pri operacijama dodele neophodno je vršiti eksplicitnu konverziju između tipova (listing 10).

```go
var i int = 5
j := i // ok, j je sada int
//var k float32 = i - ne radi
var k float32 = float32(i)
```

Listing 10 - Konverzija tipova

2. Kontrola toka programa

2.1. For

For petlja je jedini konstrukt u Go-u koji predstavlja petlju. Osnovni oblik for petlje podrazumeva tri komponente: inicijalni iskaz (uglavnom deklaracija promenljive), uslovni izraz koji se evaluira pre svake iteracije i iskaz koji se izvršava na kraju svake iteracije. Inicijalni i krajnji iskaz su opcioni i mogu se izostaviti. Iako while petlja ne postoji, ona se može predstaviti for petljom koja sadrži samo uslovni izraz.

Navedene tri komponente for petlje nije neophodno stavljati u male zagrade, dok su velike zagrade nakon toga obavezne.

Rad sa for petljom prikazan je u listingu 11.

```
//osnovna for petlja
for i := 1; i > 10; i++ {
   fmt.Println(i)
}

sum := 1
//petlja bez pocetnog i krajnjeg iskaza
for ; sum < 10 ; {
   sum += sum
}

sum = 1
//imitacija while petlje
for sum < 100 {
   sum += sum
}

//beskonacna petlja
for {

}
```

Listing 11 - For petlja

2.2. If-else

If izraz ne mora biti unutar malih zagrada, dok su velike zagrade nakon toga obavezne. Pre navođenja uslovnog izraza moguće je navesti iskaz i na taj način deklarisane promenljive vidljive su samo unutar opsega if ili bilo koje njegove else grane. Primer upotrebe if-else naredbe dat je u listingu 12.

```
func examPassed(points int, maxPoints int) bool {
   if percentage := points / maxPoints * 100; percentage >= 51 {
      return true
   } else {
      return false
   }
}
```

Listing 12 - If-else naredba

2.3. Switch

Switch naredba slična je onima u drugim programskim jezicima, uz razliku da se slučajevi proveravaju od vrha ka dnu i samo prvi pogodak na koji se naiđe biće izvršen, ne i slučajevi nakon njega, te stoga nije potrebno pisati *break* iskaze. Takođe, case izrazi ne moraju biti konstante. Primer upotrebe switch-case naredbe dat je u listingu 13.

```
func main() {
   fmt.Println("When's Saturday?")
   today := time.Now().Weekday()
   switch time.Saturday {
   case today + 0:
      fmt.Println("Today.")
   case today + 1:
      fmt.Println("Tomorrow.")
   case today + 2:
      fmt.Println("In two days.")
   default:
      fmt.Println("Too far away.")
   }
}
```

Listing 13 - Upotreba switch-case naredbe

2.4. Defer

Defer iskaz odlaže izvršavanje navedene funkcije do trenutka završetka funkcije u kojoj se naredba nalazi. Argumenti navedene funkcije se odmah evaluiraju, ali se sam poziv odlaže. Telo funkcije može sadržati više defer iskaza i oni se izvršavaju u LIFO redosledu. Primer upotrebe može se pronaći u listingu 14.

```
func main() {
   fmt.Println("counting")

   for i := 0; i < 10; i++ {
      defer fmt.Println(i)
   }

   fmt.Println("done")
}
```

Listing 14 - Upotreba defer naredbe

3. Složeni tipovi podataka

3.1. Pokazivači

Pokazivač predstavlja promenljivu čija je uloga da skladišti adresu vrednosti nekog tipa. Pokazivač tipa *T sadrži adresu memorijske lokacije u kojoj se nalazi (ili započinje) neka vrednost tipa T. Dobavljanje vrednosti na koju pokazivač pokazuje naziva se dereferenciranje pokazivača i za to služi operator *. Operator & kreira pokazivač na zadati operand. Pokazivač koji ne sadrži adresu neke vrednosti ima vrednost *nil*. Primer upotrebe pokazivača dat je u listingu 15.

```
i := 5
var p *int = &i // u pokazivač p upisuje se adresa memorijske lokacije
                // na kojoj se nalazi i
fmt.Println(p)
// output: 0xc00000a098
fmt.Println(*p)
// output: 5
*p += 5
fmt.Println(*p)
//output: 10
```

Listing 15 - Rad sa pokazivačima

3.2. Strukture

Struktura predstavlja kolekciju polja i koristi se za modelovanje stanja nekog objekta. Listing 16 prikazuje primer definisanja jedne strukture.

```
type Vertex struct {
   X, Y int
}
```

Listing 16 - Struktura Vertex

U slučaju kada se koristi pokazivač na neku strukturu, nije neophodno vršiti eksplicitno dereferenciranje pokazivača kako bi se došlo do vrednosti određenog polja, na primer *(*p).X*, dovoljno je napisati samo *p.X*.

Instanciranje i rad sa strukturama prikazan je u listingu 17.

```
func main() {
   var (
      v1 = Vertex{1, 2}
      v2 = Vertex{X: 1}
      v3 = Vertex{}
      p *Vertex  = &Vertex{1, 2}
   )
   fmt.Println(v1, p.Y, v2, v3)
   //output: {1 2} 2 {1 0} {0 0}
}
```

Listing 17 - Rad sa strukturama

Struktura u Go-u i klasa nisu ekvivalenti. Međutim, Go nudi mehanizme za programiranje u objektno orijentisanoj paradigmi.

3.3. Nizovi

Niz predstavlja strukturu fiksne dužine i čuva kolekciju elemenata istog tipa. Rad sa nizovima prikazan je u listingu 18.

```
func main() {
   var a [2]string
   a[0] = "Hello"
   a[1] = "World"
   fmt.Println(a[0], a[1])
   //output: Hello World
   fmt.Println(a)
   //output: [Hello World]

   nums := [6]int{1, 2, 3, 4}
   fmt.Println(nums)
   //output: 1 2 3 4 0 0
}
```

Listing 18 - Rad sa nizovima

3.4. Slice

Slice (dinamički niz) predstavlja sekvencu elemenata istog tipa promenljive dužine. On uvek predstavlja referencu na neki statički niz, stoga ukoliko više od jednog slice-a opisuje isti niz, izmene nad jednim slice-om biće vidljive i među ostalim slice-ovima. Slice može biti kreiran slice-ovanjem postojećeg statičkog ili dinamičkog niza ili upotrebom *make* funkcije koja vrši alokaciju memorije (zadaju se inicijalna dužina i opciono kapacitet). Dodavanje elementa u slice vrši se funkcijom *append*. Listing 19 prikazuje rad sa slice-om.

```
func main() {
   a := make([]int, 5, 5)
   printSlice("a", a)
   //output: a len=5 cap=5 [0 0 0 0 0]

   b := make([]int, 0, 5)
   printSlice("b", b)
   //output: b len=0 cap=5 []

   c := b[:2]
   printSlice("c", c)
   //output: c len=2 cap=5 [0 0]

   d := c[2:5]
   printSlice("d", d)
   //output: d len=3 cap=3 [0 0 0]

   d = append(d, 1, 2)
   printSlice("d", d)
   //output: d len=5 cap=6 [0 0 0 1 2]
}

func printSlice(s string, x []int) {
   fmt.Printf("%s len=%d cap=%d %v\n",
      s, len(x), cap(x), x)
}
```

Listing 19 - Rad sa slice-om

Iteriranje kroz niz ili slice vrši se upotrebom *range* oblika for petlje gde se pri svakoj iteraciji vraćaju dve vrednosti: trenutni indeks i vrednost na tom indeksu. Iteriranje kroz slice prikazano je na listingu 20.

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
   for i, v := range pow {
      fmt.Printf("2**%d = %d\n", i, v)
   }
}
```

Listing 20 - Iteriranje kroz slice

3.5. Mapa

Mapa predstavlja kolekciju parova ključ-vrednost. Kako bi se elementi mogli dodavati u nju, potrebno ju je prvo inicijalizovati uz pomoć funkcije *make*. Rad sa mapama prikazan je u listingu 21.

```
func main() {
   m := make(map[string]int)

   m["Answer"] = 42
   fmt.Println("The value:", m["Answer"])
   //output: The value: 42

   m["Answer"] = 48
   fmt.Println("The value:", m["Answer"])
   //output: The value: 48

   delete(m, "Answer")
   fmt.Println("The value:", m["Answer"])
   //output: The value: 0

   v, ok := m["Answer"]
   fmt.Println("The value:", v, "Present?", ok)
   //output: The value: 0 Present? false
}
```

Listing 21 - Rad sa mapama

4. Metode

Go ne podržava klase, međutim nad strukturama i imenovanim tipovima mogu se kreirati metode. Metode predstavljaju funkcije koje poseduju specijalni *receiver* argument koji se navodi između ključne reči *func* i naziva funkcije. Metoda se može definisati samo nad tipom koji se nalazi u istom paketu. Primer rada sa metodama prikazan je u listingu 22.

```go
type Vertex struct {
   X, Y float64
}

func (v Vertex) Abs() float64 {
   return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
   v := Vertex{3, 4}
   fmt.Println(v.Abs())
}
```

Listing 22 - Metode

Metode mogu biti definisane i nad pokazivačima određenog tipa, uz to da taj tip ne može i sam po sebi biti pokazivač. Kada se metoda poziva nad samom instancom nekog tipa njena vrednost se kopira i metoda radi sa tom kopijom, što uzrokuje da se nad instancom koja poziva metodu ne može vršiti nikakva izmena. U takvim situacijama pogodno je koristiti metode definisane nad pokazivačem nekog tipa, jer tada metoda radi sa originalnom vrednošću i izmene će biti vidljive. Upotreba pointer receiver-a preporučuje se i kod korišćenja velikih struktura jer poboljšava efikasnost. Primer rada sa pointer receiver-ima prikazan je u listingu 23.

```go
func (v *Vertex) Scale(f float64) {
   v.X = v.X * f
   v.Y = v.Y * f
}

func main() {
   v := Vertex{3, 4}
   v.Scale(10)
   fmt.Println(v.Abs())
//output: 50
}
```

Listing 23 - Metoda sa point receiver-om

5. Interfejsi

Interfejs predstavlja kolekciju potpisa metoda. Bilo koja vrednosti čiji tip implementira sve metode interfejsa može biti predstavljena tipom samog interfejsa. Implementacija metoda interfejsa je implicitna, podrazumeva se da tip implementira interfejs ukoliko implementira sve njegove metode. Upotreba interfejsa prikazana je u listingu 24.

```go
type Interface interface {
   Print()
}

type T struct {
   S string
}

func (t T) Print() {
   fmt.Println(t.S)
}

func main() {
   var i Interface = T{"hello"}
   i.Print()
   //output: hello
}
```

Listing 24 - Upotreba interfejsa

6. Rad sa modulima

Kako određeni projekat sazreva, količina postojećeg koda raste i samim tim može doći do slabijeg snalaženja u njemu, te ga na neki način treba modularizovati. Takođe, upotreba paketa razvijenih od strane nekog drugog u okviru projekta nije retka pojava, stoga se javlja potreba za efikasnim praćenjem svih zavisnosti koje projekat ima, kao i njihovih verzija. Navedene izazove Go od verzije 1.11 rešava upotrebom modula. Modul predstavlja kolekciju paketa koji u korenskom direktorijumu sadrže go.mod fajl. Taj fajl definiše *module path* modula, koji ujedno predstavlja i *import path* za korenski direktorijum, kao i sve zavisnosti koje dati modul ima prema drugim modulima (moduli potrebni za uspešan build). Primer go.mod fajla nalazi se u listingu 25.

```
module example.com/mymodule

go 1.17

require (
    example.com/othermodule v1.2.3
    example.com/thismodule v1.2.3
    example.com/thatmodule v1.2.3
)
```

Listing 25 - go.mod fajl

Neke od osnovnih naredbi za rad sa modulima su:

- *go mod init naziv_modula* - kreira novi modul sa zadatim nazivom (kreira se prazan go.mod fajl u trenutnom direktorijumu)
- *go get naziv_paketa@verzija* - dobavlja modul kom zadati paket pripada i navodi ga u go.mod fajlu (ako se kao verzija navede reč *none*, dati modul će biti uklonjen)
- *go mod tidy* - vrši sinhronizaciju zavisnosti u source kodu i u go.mod fajlu, tako što dodaje nedostajuće zavisnosti ili briše one koji nisu u upotrebi
- *go list -m all* - izlistava trenutni modul i sve njegove zavisnosti

7. Konkurentno programiranje

Jedna od osnovnih prednosti Go programskog jezika je ta da nudi direktnu podršku za konkurentno programiranje, upotrebom goroutine-a i channel-a. Više o konkurentnom programiranju i konkurentnom programiranju u Go-u možete pronaći u dodatnim materijalima.

8. Dodatni materijali

- https://golangbyexample.com/
- https://learnxinyminutes.com/docs/go/
- https://gobyexample.com/
- https://golangbot.com/learn-golang-series/
- https://www.youtube.com/watch?v=CF9S4QZuV30
- https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/
- https://www.golangprograms.com/go-language/concurrency.html
- https://vimeo.com/49718712