Programmazione Teoria

Indice

1 Tipi base in Go

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64 *uintptr*
- byte //alias di uint8
- rune //alias di int32
- float32 float64
- complex64 complex128

int e uint sono "implementation dependent" (occupano 32 o 64 bit in base all'architettura della macchina in cui si esegue il codice).

1.1 Strong typing

Un linguaggio strong typing non cambia il tipo delle variabili implicitamente (come Go). Per esempio se divido due numeri interi (int), otterrò sempre un numero intero e non un numero decimale (float).

1.2 Cast

Cambio del tipo di variabili esplicitamente. Per esempio se voglio approssimare un numero con la virgola (float) devo fare un casting.

```
var decimale float64 = 20.4
var intero int = int(decimale) //il valore di intero è 20
```

1.3 Coercion

Cambio del tipo di variabili implicitamente. Per esempio se voglio sommare un numero con la virgola con numero intero il compilatore di un linguaggio *weak typing* cambierà il tipo della variabile intera.

2 Variabili

Le variabili sono un contenitore, identificati da un nome univoco, di un qualsiasi valore.

2.1 La vita di una variabile

1. **Dichiarazione**:
   - Keyword: var
   - Identificatore: nome della variabile
   - tipo: tipo della variabile
2. **Definizione** (inizializzazione o allocazione)
3. **Utilizzo**
4. **Rilascio**

2.2 Dichiarazione e definizione breve

Per evitare di scrivere la keyword `var` e il tipo della variabile, in Go si può scrivere `nomeVariabile := valore`

```
numeroIntero := 22              //tipo int
numeroFloat := 20.19            //tipo float64
numeroCarattere := 'a'          //tipo int32
stringa := "Hello World!"       //tipo string
```

2.3 Scope

Parte del codice in cui una variabile è accessibile.

Quando si dichiarano le variabili fuori da tutti le funzioni il loro scope è **globale**, ovvero accessibile in tutto il codice.

2.4 Zero value

Quando di dichiarano una variabile senza esplicitare il valore, il valore sarà:

Tipo	Valore
Integer	0
Floating point	0.0
Boolean	false
String	“”
Altri tipi	nil

2.5 Costanti

Le costanti sono un contenitore, identificati da un nome univoco, di un qualsiasi valore immutabile.

È buona norma scrivere le constanti in maiuscolo e fuori dalle funzioni.

Di solito si utilizzano per la dimensione per gli array e per le costanti matematiche.

```
const NOMECOSTANTE tipo = valore
```

2.6 Swap

Scambio del contenuto tra due variabili.

```
var a int
a = 1
var b int
b = 2
a, b = b, a
//adesso il valore di a è 2
//e il valore di b è 1
```

3 Operatori in Go

3.1 Operatori binari

Binari	Descrizione
*	Moltiplicazione
/	Divisione
%	Calcolo del resto della divisione
|	OR binario
^	XOR binario
&	AND binario
&^	AND NOT binario
+	Somma
-	Sottrazione
«	Shift a sinistra
»	Shift a destra
==	Verifica ugguaglianza
!=	Verifica diversità
<	Minore
<=	Minore o uguale
>	Maggiore
>=	Maggiore o uguale
&&	AND logico
||	OR logico

3.2 Operatori unari

Unari	Descrizione
++	Incrementa di 1
–	Sottrae di 1
+	Più unario
-	Meno unario
!	Not
ˆ	Complemento a 1
*	Valore dell'indirizzo
&	Indirizzo
<-	Ricevitore

4 Errori

- **Lessicale**: uso di parole chiave del linguaggio non esistenti o scritte in maniera errata
- **Sintattico**: uso di parole chiave del linguaggio scritte correttamente ma utilizzate in maniera errata
- **Semantico** (*o logico*): fornisce un output incoerente o si comporta in un modo inaspettato
- **Estetico** (*o pragmatico*): scelta di andare a capo errata, indentazione errata (distribuzione del codice) e scelta dei nomi degli identificatori poco chiara

5 If

Per controllare il flusso del programma si può usare il costrutto **if/else**. **else** è facoltativo. Si possono anche concatenare diversi **if/else**.

```
if condizione {
    //se la condizione è vera esegue queste istruzioni
} else {
    //altrimenti se la condizione è falsa esegue queste istruzioni
}

if condizione {
    //sequenza di istruzioni se la condizione è vera
}

if condizione {
    //sequenza di istruzioni se la condizione è vera
} else if condizione {
    //sequenza di istruzioni se la condizione è vera
} else {
    //altrimenti se le condizioni sono false esegue queste istruzioni
}
```

6 For

Il costrutto `for` cicla una sequenza di istruzioni delimitate da `{}` finché la condizione è valida.

Ogni ripetizione è chiamata iterazione.

6.1 Unaria

```
for condizione {
    //se la condizione è vera esegue queste istruzioni
}
```

6.2 Ternaria

```
for inizializzazione; condizione; aggiornamento {   //esegue l'inizializzazione solo una volta
    //se la condizione è vera esegue queste istruzioni
```

```
    //alla fine delle istruzioni precedenti esegue l'aggiornamento
}
```

6.3 Zeraria

```
for {
    //se la condizione è vera esegue queste istruzioni
}
```

Attenzione! Con la forma zeraria è necessario creare una condizione nella quale sia possibile usare `break` per uscire dal ciclo altrimenti il ciclo sarà infinito.

`continue` invece permette di ingnorare le seguenti istruzioni e di passare alla successiva iterazione.

6.4 Range

Utile quando si deve scorrere su tutta una stringha, array, slice e mappa.

```
for indice, valore := range variabile {
    //sequenza di istruzioni
}
```

7 Sottoprogramma

Parte di codice che svolge un compito specifico. Può accettare parametri in input e restituire parametri in output.

I sottoprogrammi sono utili per rendere il codice modulare e per aumetare la leggibilità.

Nel caso in cui non ci sia necessità di passare i parametri effettivi, i parametri formali si possono omettere. Nel caso in cui non ci sia necessità di restituire alcun valore è possibile omettere il tipo del valore restituito.

```
func nomeFunzione(nomeParametriFormali tipoParametriFormali) tipoValoreRestituito {
    //sequenza di istruzioni
    return nomeVariabile
}
```

Eventualmente è possibile dare un nome alla variabile restituita. In quel caso si può omettere il nome della variabile nel `return`

```
func nomeFunzione(nomeParametriFormali tipoParametriFormali) (nomeVariabile tipoValoreRestituito) {
    //sequenza di istruzioni
    return
}
```

Attenzione! I parametri sono passati per **copia**.

7.1 Utilizzo

Per usare i sottoprogrammi è neccessario fare una chiamata alla funzione desiderata:

```
variabileTipo = nomeFunzione(nomeParametriEffettivi) //chiamata alla funzione "nomeFunzione"
//dando "nomeParametriEffettivi" come input
//e assegno a "variabileTipo" il valore restituito da "nomeFunzione"
```

Nel caso la funzione non restituisca alcun valore:

```
nomeFunzione(nomeParametriEffettivi) //chiamata alla funzione "nomeFunzione"
//dando "nomeParametriEffettivi" come input
```

7.2 Scope

camelCase: (scrivere parole attaccate con ogni iniziale maiuscola, tranne la prima) rende il sottoprogramma privato (accessibile solo nello stesso file)

PascalCase: (scrivere parole attaccate con ogni iniziale maiuscola) rende il sottoprogramma Pubblico (accessibile in tutti i file in cui si sia importato il package)

8 Selezione multi-aria: Switch - case

Per controllare il flusso del programma si può usare i costrutto **switch-case**. Esegue delle istruzioni in base all'espressione testa. Se nessun caso è valido allora sarà eseguita la sezione `default`.

Inserire la sezione `default` è opzionale.

```
switch espressioneTesta {
case espressione1a, espressione1b:
    //esegue queste istruzioni se espressioneTesta == espressione1a oppure
    //se espressioneTesta == espressione1b
case espressione2:
    //esegue queste istruzioni se espressioneTesta == espressione2
default:
    //esegue queste istruzioni se i casi sopra non sono validi
}
```

Se non specificata, l'espressione testa è `true`

```
switch { //switch true
case espressioneBooleana1:
    //esegue queste istruzioni se espressioneBooleana1 è true
case <espressione booleana 2>:
    //esegue queste istruzioni se espressioneBooleana2 è true
default:
    //esegue queste istruzioni se i casi sopra non sono true
}
```

Attenzione! Quando si entra in un case poi si esce dal costrutto **switch-case**.

9 Packages e funzioni utili

9.1 fmt

Package `"fmt"`, lo usamo per stampare e per scannerizzare.

È possibile andare a capo con `\n`

È possibile fare una tabulazione con `\t`

Scopri il package fmt

9.1.1 Print

Funzione che stampa il contenuto tra parentesi.

```
fmt.Print()
```

9.1.2 Println

Funzione che stampa il contenuto tra parentesi poi va a capo.

```
fmt.Println()
```

9.1.3 Printf

Funzione che stampa il contenuto tra parentesi, si possono utilizzare diversi verbi (specificatori di formato).

```
fmt.Printf("Testo %SpecificatoreFormato", nomeVariabile)
```

Principali verbi per Integer	Descrizione
%d	base 10
%c	carattere unicode
%b	base 2
%x	base 16
%o	base 8

Principali verbi per String e Slice	Descrizione
%s	stringa o slice

Principali verbi per Float	Descrizione
%f	numeri decimali

Principali verbi per Bool	Descrizione
%t	true o false

9.2 math

9.2.1 math/rand

9.2.1.1 Int

Genera un numero intero pseudorandomico.

```
var n int = rand.Int()
```

9.2.1.2 Intn

Genera un numero intero pseudorandomico compreso tra due valori.

```
var n int = rand.Intn(massimo-minimo) + minimo
```

9.2.1.3 Seed

Seed modifica l'origine del calcolo numero pseudorandomico. Se non viene dato alcun valore il valore sarà 1.

```
rand.Seed(valore)
```

9.3 os

9.3.1 Args

Restituisce un array di string letti come argomenti da linea di comando, in prima posizione ci sarà il nome del programma.

```
go run programma.go elemento1 elemento2 elemento3
```

9.3.2 Stdin

```
var s []string = os.Args[1:]
```

9.4 bufio

9.4.1 NewScanner

9.5 time

9.5.1 Now

9.5.1.1 UnixNano

10 Ascii

128 caratteri da 0 a 127 rasppresentabili con 1 byte.

10.1 Tipo byte

Alias uint8, usato per rappresentare i caratteri ASCII. Occupa 1 byte.

```
var carattere byte = 65
```

11 Unicode

I primi 128 “grafemi” sono gli stessi dell ASCII e sono necessari fino a 4 byte.

11.1 Rune

Alias di int32, usato per rappresentare i grafemi Unicode. Occupa 4 byte.

```
var grafema rune = 8984
```

12 Stringhe

Sequenza di `byte`, ovvero un array di byte.

13 Tipi composti in Go

- array
- puntatori
- slice
- struct
- map
- *tipi funzione*

Per sapere la lunghezza di: stringhe, array, slice e mappe si usa la funzione `len()`

13.1 Array

Sequenza di variabili dello stesso tipo di dimensione statica (non cambia).

```
var nomeArray [lunghezza]tipo
```

```
var numeri1 [4]int
numeri1 = [4]int{1, 2, 3, 4}
var numeri2 [4]int = [4]int{5, 6, 7, 8}
numeri3 := [4]int{9, 10, 11, 12}
```

13.2 Puntatori

Un puntatore contiene l’indirizzo di una variabile, dall’indirizzo possiamo ricavare il contenuto.

`&` referenziazzione (l’indirizzo).

`*` dereferenziazzione o indirezione (valore).

```
var nomePuntatore *tipo       dichiarazione
nomePuntatore = &variabile    referenziazione
variabile = *nomePuntatore    dereferenziazzione
```

```
package main

import "fmt"

func main(){
    var b int = 20
    var a *int
    a = &b
    fmt.Println("L'indirizzo di 'b' è ", a)
    fmt.Println("Il valore di 'b' è ", *a)
    sommo2(a) //Modifico 'a'
    fmt.Println("Il valore di 'b' dopo la modifica ad 'a' è ", b)
}

/*
```

```
/*
Sto passando per copia il puntatore 'a', ma dato che 'a' ha come indirizzo l'indirizzo di 'b',
è come se stessi modificando 'b', quindi posso omettere il return
*/
func sommo2(n *int){
    *n = *n + 2
}
```

13.3 Slice

Sequenza di variabili dello stesso tipo di dimensione dinamica (può cambiare).

Una slice ha tre componeni:

- puntatore: punta agli elementi dell'array
- lunghezza: è il numero massimo di elementi della slice
- capacità: rappresenta la dimensione massima fino alla quale la slice può espandersi senza riallocarsi

`var nomeSlice []tipo`

Per inizializzare una slice si usa la funzione `make()`

`nomeSlice = make([]tipo, lunghezza, capacità)`

Per aumentare la lunghezza di 1 e aggiungere un valore si usa la funzione `append()`

`nomeSlice = append(nomeSlice, valore)`

Per sapere la dimensione della capacità della slice si usa la funzione `cap()`

```
var s []int
s = make([]int, 4, 20)
s = append(s, 2)
```

13.3.1 Subslice

Una slice ha una referenziazione a un array. Quindi non è difficile crearli partendo da un array.

Per creare le subslice (parti di array o slice) si usa il nome della variabile e la posizione iniziale inclusa e la posizione finale esclusa separate da `:` e comprese da `[]`.

`nomeArrtayOSlice[posIni:posFin]`

Se si omette la posizione iniziale essa coinciderà con la posizione iniziale dell'array o slice, mentre se si omette la posizione finale essa coinciderà con la posizione finale dell'array o slice.

```
var array [5]int = [5]int{1, 2, 3, 4, 5}
var subSlice0 []int = array[2:4]    //subslice da un array dalla posizione 2 a 3

var slice []int = []int{1, 2, 3, 4, 5}
var subSlice1 []int = [1:3]         //subslice da una slice dalla posizione 1 a 2
var subSlice2 []int = [1:]          //subslice da una slice dalla posizione 1 fino alla posizione fine
var subSlice3 []int = [:3]          //subslice da una slice dalla posizione 0 fino alla posizione 2
var subSlice4 []int = [:]           //subslice da una slice dalla posizione 0 fino alla posizione finale
```

Attenzione! Quando si crea una subslice si copiano anche gli indirizzi, quindi è possibile modificare i valori dell'array o della slice iniziale senza volere.

13.3.2 Slice di slice

In Go possiamo creare delle slice di slices, ovvero slice multidimensionali.

```
package main

import "fmt"

const RIGHE int = 4
const COLONNE int = 5
```

```go
func main() {
    var matrice [][]int            //dichiaro matrice (slice di slice)
    matrice = make([][]int, RIGHE) //alloco la matrice, e do la lunghezza (numero di righe)
    //è possibile aggiungere la capacità

    matrice = creaMatriceForRange(matrice)
    matrice = caricaMatriceForRange(matrice)
    stampaMatriceForRange(matrice)
}

func creaMatriceForRange(m [][]int) [][]int {
    for i, _ := range m {
        m[i] = make([]int, COLONNE) //creo le colonne, è possibile aggiungere la capacità
    }
    return m
}

func caricaMatriceForRange(m [][]int) [][]int {
    for i, _ := range m { //righe
        for k, _ := range m[i] { //colonne
            m[i][k] = i * k
        }
    }
    return m
}

func stampaMatriceForRange(m [][]int) {
    for i, _ := range m { //righe
        for k, _ := range m[i] { //colonne
            fmt.Printf("%4d", m[i][k])
        }
        fmt.Println()
    }
}

func creaMatriceFor(m [][]int) [][]int {
    for i := 0; i < RIGHE; i++ {
        m[i] = make([]int, COLONNE) //creo le colonne
    }
    return m
}

func caricaMatriceFor(m [][]int) [][]int {
    for i := 0; i < RIGHE; i++ { //righe
        for k := 0; k < len(m[i]); k++ { //colonne
            m[i][k] = i * k
        }
    }
    return m
}

func stampaMatriceFor(m [][]int) {
    for i := 0; i < RIGHE; i++ { //righe
        for k := 0; k < len(m[i]); k++ { //colonne
            fmt.Printf("%4d ", m[i][k])
        }
    }
}
```

13.4 Map

Array associativi (come un dizionionario: ad ogni voce corrisponde una definizione), associo a una chiave un valore. Per aggiungere una chiave e il suo valore si deve prima allocare la mappa. È possibile eliminare elementi della mappa con la funzione `delete()`

```
var nomeMappa map[tipoChiave]tipoValore          //dichiarazione
nomeMappa = make(map[tipoChiave]tipoValore)      //allocazione
nomeMappa[chiave] = valore                       //aggiungo elemento
delete(nomeMappa, chiave)                        //elimino elemeto
```

```
var m map[string]int
m = make(map[string]int)
m["a"] = 2
delete(m, "a")
```

13.5 Struct

Le struct ci permettono di raggruppare variabili, anche di diversi tipi.

Si possono dichiarare fuori dalle funzioni per creare nuovi tipi:

```
type nomeStruct struct{
    nomeCampo1 tipo
    nomeCampo2 tipo
}
```

```
type Studente struct{
    Nome, Cognome string
    matricola int
}
```

Oppure dentro le funzioni.

```
var nomeVariabile1 struct{
    nomeCampo1 tipo
    nomeCampo2 tipo
}
```

```
var Studente struct{
    Nome, Cognome string
    matricola int
}
```