GOLANG

Visibilità (Exported - Unexported)

In Go un nome viene esportato (exported, pubblico), ossia è visibile al di fuori del package se comincia con la lettera maiuscola. Quando si importa un package, ci si può solo riferire a i nomi "exported" (pubblici). Qualsiasi nome "unexported" (privato) non è accessibile al di fuori del package.

Variabili

La dichiarazione `var` crea una variabile di un determinato tipo, le assegna un nome ed un valore iniziale. Ogni dichiarazione ha la seguente forma.

```
var name type = expression
```

Sia il `type` che `= expression` possono essere omessi, ma non entrambi.

1. Se il tipo è omesso, allora viene dedotto (type inference) dall'espressione inizializzatrice.
2. Se invece è omessa l'espressione inizializzatrice allora il valore iniziale della variabile è il valore zero per quel tipo.

```
var str, b = "hello", true  // caso 1, type inferece
var i = 27                  // caso 1, type inference

var i int           // caso 2, inizializzato con valore zero per tipo int
```

Con lo statement `var` è possibile dichiarare anche una lista di variabili.

```
var i, y int
var x string
```

Anche in questo caso la dichiarazione di una variabile può includere l'inizializzazione, una per variabile

```
var i, j int = 1, 2
```

Short variables declaration

Solo all'interno di una funzione è possibile utilizzare (molto consigliato) l'operatore `:=` (short variable declaration) al posto della dichiarazione con `var` (vedi sopra).

```
name := expression
```

La *short variable declaration* consiste in dichiarazione + inizializzazione con deduzione del tipo (type inference).

> **NB:** una *short variable declaration* non necessariamente dichiara tutte le variabile poste nel lato sinistro. Se **alcune** di queste sono già state dichiarate nello stesso blocco lessicale, allora la *short variable declaration* agisce come **assegnamento**.

```
func main() {
    i:= 8
    i, k:= 7,9  // i assegnamento, k dichiarazione + inizializzazione
}
```

Tempo di vita di una variabile

Il tempo di vita di una variablie è l'intervallo di tempo durante il quale esiste mentre il programma è in esecuzione.

Il tempo di vita di una variabile a livello di package è **l'intera esecuzione del programma**.

Una variabile locale invece ha un tempo di vita dinamico: una nuova istanza è creata ogni qual volta lo statement di dichiarazione viene eseguito, e **la variabile vive fino a che non diventa irragiungibile**.

Uno degli effeti di questa caratteristica in Go sono le *closures*.„

Dichiarazione di Tipo

Il tipo di una variabile (o espressione) definisce le caratteristiche dei valori che può assumere, come size, operazioni ecc.

Una dichiarazione di tipo definisce un nuovo **tipo** con nome (**named type**), che ha lo stesso tipo sottostante di un tipo esistente.

Il nuovo tipo fornisce un modo per separare usi differenti del tipo sottostante in modo che non possano essere mischiati se non intenzionalmente.

```
type name underlyingType
```

Questa dichiarazione molto spesso avviene a livello di package.

Type Inference

Quando si dichiara una variablie senza specificarne esplicitamente il tipo (sia usando la sintassi `:=` che `var`), il tipo della variabile è dedotto (inferred) dal valore dell'espressione.

Se l'espressione ha un **tipo**, la nuova variabile assume quel tipo.

```
var i int

j := i      // j è un int
var k = i   // k è un int
```

Se invece il valore dell'espressione è una **costante numerica** senza tipo, la nuova variabile potrebbe assumere come tipo un `int`, `float64`, o `complex128` a seconda della precisione della costante

```
i := 42             // int
f := 3.142          // float64
g := 0.867 + 0.5i   // complex128
```

Valori zero

Variabili dichiarate senza un inizializzatore esplicito sono automaticamente inizializzate con il proprio "valore zero", il quale è :

`0` per i tipi numerici

`false` per il tipo booleano

`""` stringa vuota per le stringhe

`nil` per interfacce e tipi riferimento cioè *slice, pointers, map, channel, function*

Il valore zero di un tipo aggregato come *array e struct* ha il valore zero di tutti i suoi elementi o campi.

> **NB**: Il meccanismo "zero values" garantisce che una variabile contenga sempre un valore ben definito del suo tipo.

Tipi base

I tipi di base del linguaggio GO sono

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // alias per uint8

rune // alias per int32
     // rappresenta un Unicode code point

float32 float64
```

```
complex64 complex128
```

I tipi `int` , `uint` , e `uintptr` occupano di solito 32 bits su sitemi a 32-bit e 64 in sistemi a 64-bit.

Quando si vuole usare un valore intero, è consigliato usare `int` a meno che ci sia una ragione specifica per usare un tipo intero con size specifica o senza segno.

Conversione di tipo

L'espressione `T(v)` converte il valore `v` al tipo `T`

```
i := 27
f := float64(i)
u := uint(f)
```

A differenza del C, in Go l'assegnamento tra elementi di tipi differenti richiede una **conversione esplicita**.

Per ogni tipo `T` , è sempre possibile un'operazione di conversione `T(v)` che converte il il valore `v` al tipo `T` . Una conversione da un tipo ad un altro è possibile se:

- `T` e `v` hanno lo stesso *tipo sottostante*
- `T` e `v` sono di tipo *unamed pointer* che punta ad una variabile dello stesso *tipo sottostante*

Questa conversione cambia cambia il tipo ma non la rappresentazione del valore. Se `v` è assegnabile a `T` , la conversione è permessa ma è un'operazione ridondante.

Le conversione sono permesse anche tra:

- tipi numerici
- stringhe
- alcuni tipi di slices

Stringhe

Intro

Per riferirsi ai caratteri in modo non ambiguo, ogni carattere è associato ad un numero, chiamato **code point**. I caratteri sono salvati nel computer come uno o più bytes.

Una codifica di caratteri (character encoding) fornisce una chiave per interpretare il codice. È un insieme di mappature tra bytes nel computer e caratteri nell'insieme di caratteri. Senza la chiave, i dati sono spazzatura.

Quando viene inserito del testo usando una tastiera (o altro), la codifica di caratteri mappa i caratteri inseriti in specifici byte nella memoria del computer e quindi per visualizzare il testo converte nuovamente i byte in caratteri.

Una stringa è una sequenzaimmutabile di bytes (uint8). Le stringhe possono contenere dati arbitrari, inclusi bytes con valore 0, ma solitamente contengono testo leggibile.

Una stringa contiene una array di bytes che una volta creato, è immutabile. Al contrario, gli elementi di una slice di byte possono essere modificati liberamente.

Una *stringa* può essere convertita in uno *slice di byte* e viceversa.

Le stringhe sono interpretate come sequenze codificate in UTF-8 di Unicode code points (runes).

La funzione `len` del pkg `builtin` restituisce il # di bytes (non runes) in una stringa, e l'operazione di indicizzazione `str[i]` restituisce l'i-esimo byte della stringa.

```
str := "samuele"

fmt.Println(str[0]) // stampa 115
fmt.Println(string(str[0])) //stampa "s"
```

L'operazione di *substring* `str[i:j]` produce una nuova stringa contenente i bytes della stringa originale partendo dall'indice `i` fino all'indice `j` escluso.

```
str := "samuele"

str[0:3]    // produce la stringa "sam"
```

L'operatore `+` crea una nuova stringa concatenandole

```
str1 := "ciao"
str2 := " "
str3 := "mondo"

str1 + str2 + str3  // produce la stringa "hello world"
```

Le stringhe possono essere confrontate con gli operatori di comparazione come `==`, `<`. La comparazione viene fatta byte per byte.

```
str := "samuele"

fmt.Println(str[0]) // stampa 115
fmt.Println(str[1]) // stampa 97
fmt.Println(str[1] < str[0])    // stampa true
```

Stringhe letterali

Una valore di tipo stringa può essere scritto come una *stringa letterale*, una sequenza di bytes racchiusi da doppi apici `"Hello"`.

Poichè i file sorgenti in Go sono sempre codificati in UTF-8 e le stringhe in GO sono convenzionalmente interpretate come UTF-8, e possibile includere Unicode code point (rune) nelle stringhe letterali.

In una stringa letterale con doppi apici, le sequenze di escape che iniziano con la barra rovesciata `\` possono essere utilizzate per inserire valori di byte nella stringa. Un insieme di escapes gestisce codici di controllo ASCII come *newline*, *carriage return* e *tab*.

```
\a  // alert o bell
\b  // backspace
\f  // form feed
\n  // new line
\r  // carriage return
\t  // tab
\v  // vartical tab
\'  // single quote (solo su rune letterali)
\"  // double quote (solo con letterali stringa)
\\  // backslash
```

Una *stringa letterale raw* è scritta tra backquotes ``` `` ``` invece che double quotes `""`. In una stringa letterale raw le sequenze di escape non sono processate

```
str := ` Ciao
Mondo
`
// stampa: Ciao
//        Mondo
```

Le stringe letterali raw anche usate per scrivere espressioni regolari, in quanto tendono ad avere un gran numero di backslashes.

Costanti

Le costanti sono dichiarate come le variabili, ma con la keyword `const` e definiscono un valore costante il quale previene accidentali modifiche durante l'esecuzione del programma.

```
const name type = expression
```

Il tipo sottostante di ogni costante è un tipo base: *booleano, stinga* o *numerico*, di conseguenza una costante può assumere solo questi tipi.

Le costanti non possono essere dichiarati usando la sintassi `:=`.

Come per le variabili, una sequenza di costanti può apparire in una dichiarazione. Può essere appropriato per un gruppo di valori correlati

```
const (
    pi = 3.14159
    e  = 3.14159
)
```

Quando una sequenza di costanti è dichiarata in gruppo, l'espressione alla destra dell' `=` può essere omessa per tutti tranne per la prima del gruppo, il che implica che l'espressione precedente e il suo tipo vengono usate nuovamente

```
const (
    a = 1
    b       // b == 1
    c = 2
    c       // c == 2
)
```

Il generatore di costanti iota

Una dichiarazione di costante può fare uso del *generatore di costante* `iota`, il quale viene usato per creare una sequenza di valori correlati senza specificare esplicitamente ciascuno di essi. In una dichiarazione di costante il valore di `iota` inizia da `0` e viene incrementato di uno per ogni elemento in sequenza.

```
type Weekday int

const (
    Sunday Weekday = iota       // valore 0
    Monday                      // valore 1
    Tuesday                     // valore 2
    Wednesday                   // valore 3
    Thirsday                    // valore 4
    Friday                      // valore 5
    Saturday                    // valore 6
)
```

Costanti senza tipo

Anche se una costante può assumere qualsiasi tipo di base (numeric , bool, string), molte costanti sono associate ad un tipo particolare. Il compilatore rappresenta questa costanti con una precisione numerica maggiore rispetto ai valori dei tipi di base. Si può. assumere una precisione di 256 bit.

Ci sono sei tipologie di "uncommitted constants":

- Untyped boolean
- Untyped integer
- untyped rune
- untyped floating-point

- Untyped complex
- untyped string

Rimandando l'assunzione di un tipo, le costanti non tipizzate non solo mantengono la maggiore precisione piu a lungo, ma possono partecipare a molte più espressioni rispetto alle costanti con tipo s**enza richiedere conversioni**.

For statements

Go ha solo un costrutto per i cicli, il ciclo `for`. Il ciclo `for` di base ha **tre** componenti separati da `;`

- *init statement*: viene eseguito prima della prima iterazione
- *condition expression*: viene valutato prima di ogni iterazione
- *post statement*: viene esegutio alla fine di ogni iterazione

L' *init statement* solitamente è una *short variable declaration*, e la variablie dichiarata è visibile solo nello scope del ciclo `for`. Non ci sono parentesi che circondano i 3 componenti del ciclo `for`.

```
for i := 0 ; i < 10; i++ {...}
```

L' *init e post statements* sono opzionali. Omettendoli si ottiene un ciclo while.

```
sum := 1
for sum < 100 {
    sum += sum
}
```

Se si omette anche la "condition expression" allora il ciclo va all'infinito.

```
for {...}
```

Range

Un `for` statement con una clausola `range` itera attraverso tutt le entries di:

- array
- slice
- map
- valori ricevuti da un channel

L'espressione alla destra in una clausola `range` è chiamata *range expression*, e può essere un *array*, un puntatore ad un **array*, *slice*, *string*, *map* o *channel* che consente "receive operations".

```
sli := []int{1, 2, 3, 4, 5}

// range su slice
for i, v := range sli {
    fmt.Println(i, v)
}

// range su map
// range su array
// range su *array
// range su string
// range su channel
```

Per ogni entry la clausala `range` assegna i **valori di iterazione** alle **variabile di iterazione** corrispondenti se presenti e quindi esegue il blocco.

Per ogni iterazione, i valori di iterazione sono prodotti seguendo le rispettive variablile di iterazione secondo la seguente tabella

Range expression	Tipo	1st valore	2nd valore
array o slice	a [n]E, *[n]E, []E	Index i int	a[i] E
String	s string	Index i int	s[i] rune
map	m map[K]V	Key k K	m[k] V
Channel	c chan E, <-chan	elemento e E	

Blank Identifier

il *blank identifier* `_` può essere assegnato o dichiarato con qualsiasi valore di qualsiasi tipo, e il valore viene scartato in modo innocuo (in go ogni variabile cdeve essere usato altrimenti viene segnato un errore di compilazione). Si tratta di un valore di sola scrittura da utilizzare come segnaposto dove è necessaria una variabile ma il valore non ci interessa.

```
sli := []int{1, 2, 3, 4}
for _, v := range sli { // nessun errore di compilazione
    fmt.Println(v)
}
```

If statemets

In Go, come per il ciclo `for`, nello statement `if` l'espressione non ha bisogno di essere racchiusa da parentesi rotonde, ma per il corpo sono richieste le parentesi graffe.

```
if condition {...}
```

Inoltre l' `if` statement può iniziare con un breve statement che viene eseguito prima della condizione. Le variabili dichiarate dallo statement sono visibili solo nello scope dell' `if`.

```
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}
```

Le variabile dichiarate all'interno di un `if` short statement sono anche visibili all'interno dei blocchi `else`.

```
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    } else {
        fmt.Println(v)  // v è visibile
    }
    return lim
}
```

Switch statements

Uno `switch` statement è il modo piu rapido per scrivere una sequenza di `if - else` statements. Ci sono 2 forme di switch: **expression switches** e **type switches**.

Expression switches

In un expression switch i cases contengono espressioni che sono comparate con il valore dell'espressione dello switch. **Viene eseguito il primo caso il cui valore sia uguale alla condizione**.

Gli switch cases vengono valutati dall'alto verso il basso, fermandosi quando viene eseguito il primo caso. Se nessun case matcha con l'espressione allora, se presente, viene eseguito il default case.

A differenza di altri linguaggi, in Go:

1. non è necessario usare lo statement `break` in quanto è automaticamente inserito dal linguaggio alla fine di ogni case.
2. gli *switch cases* non devono necessariamente essere costanti
3. i valori coinvolti negli switch cases non devono necessariamente essere interi.

La switch expression può essere preceduta da uno statement semplice.

```
// switch preceduto da statement semplice. i nello scope dello switch
switch i := expression; i {
    case value1:    // se i == value1 tutto il resto non viene valutato
```

```
        ...
    case value2:
        ...
    default:
        ...
}

// oppure si può evitare l'init statement, e inserire solo la switch
expression
i := "Hello"
switch i {
    case value1:  // se i == value1 tutto il resto non viene valutato
        ...
    case value2:
        ...
    default:
        ...
}
```

Uno `switch` senza una condizione è lo stesso di uno `switch true` . È un modo pulito per scrivere catene di `if-then-else`

```
switch {
    case value1 < value:
        ...
    case value2 == value:
        ...
    default:
        ...
}
```

è possibile raggruppare più casi per uno stesso comportamento con una lista separata da `,`

```
c := '?'
switch c {
    case ' ', '?', '&', '=':
        return true
}
```

Quando un case viene eseguito (quindi matcha con la switch expression) si esce dallo switch (break implicito). Se invece si vuol far passare il controllo al case successivo, si può usare la keyword `fallthrough` alla fine del corpo del case.

```
v := 100
switch v {
    case 100:           // invece di stampare 100 e uscire con il break
implicito
        fmt.Println(100)    // con fallthrough il flusso dell'esecuzione
passa al case
        fallthrough     // successivo
    case 1:
        fmt.Println(1)
        fallthrough
    default:
        fmt.println("default")

}
```

Type switches

Un *type switch* è un costrutto che permette diversi **assertion types** in serie. Si tratta di un regolare *switch statement* ma i casi in un *type switch* specificano tipi (non valori), e questi sono confrontati con il tipo del valore contenuto nell'interfaccia specificata.

```
switch v := i.(type) {
case T:
    // qui v ha tipo T
case S:
    // qui v ha tipo S
default:
    // nessun match; qui v ha lo stesso tipo di i
}

func do(i interface{}) {
    switch v := i.(type) {
    case int:
        fmt.Printf("is an int")
    case string:
        fmt.Printf("%q is %v bytes long\n", v, len(v))
    default:
        fmt.Printf("I don't know about type %T!\n", v)
    }
}

func main() {
    do(21)      // Twice 21 is 42
    do("hello") // "hello" is 5 bytes long
    do(true)    // I don't know about type bool!
}
```

Defer

Una dichiarazione di `defer` rimanda l'esecuzione di una funzione fino a quando la funzione che la contiene non ritorna. L'argomento di una dichiarazione `defer` viene valutata immediatamente, ma la funzione viene chiamata solo al ritorno della funzione contenitrice.

```
func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
}

// stampa: hello
//     world
```

Le chiamate a funzione `defer` sono inserita in uno stack (pila). Quando la funzione che contiene gli statement `defer` ritorna, le sue chiamate posticipate sono eseguito in ordine Last-In-First-Out, quindi al contrario dell'ordine di dichiarazione nel corpo della funzione.

```
func main() {
    defer fmt.Println("1")
    defer fmt.Println("2")
    defer fmt.Println("3")
    defer fmt.Println("4")

    fmt.Println("hello defer")
}

// stampa: hello defer
//     4
//     3
//     2
//     1
```

Pointers types

Un puntatore contiene l'indirizzo in memoria di una variabile. Il tipo `*T` è un puntatore ad un valore di tipo `T`. Il valore zero di un puntatore è `nil`.

```
var p *int
```

L'operatore `&` applicato ad un valore restituisce il suo indirizzo in memoria

```
i := 27
p := &i
```

L'operatore `*` si chiama operatore di dereferenziazion/indirezione ed applicato ad un puntatore restituisce l'oggetto puntato

```
fmt.Println(*p) // leggo i attraverso il puntatore p
*p = 23     // metto i attraverso il puntatore p
```

Arrays types

Un array è una **sequenza fissa** di zero o più elementi di **un particolare tipo**. A causa della dimensione fissa, in Go gli array sono raramente utilizzati a favore degli **slices**.

Il tipo `[n]T` è un array di `n` valori di tipo `T`. L'espressione

```
var a [10]int   // array di 10 interi, inizializzati con il valore zero per il tipo
        // il tipo è [10]int
```

dichiara una variabile `a` come un array di dieci interi

> **NB**: La lunghezza di un array è parte del suo tipo, perciò gli array non possono essere ridimensionati.

Per accedere agli elementi di un array si usa la consueta notazione di subscripting

```
fmt.Println(a[0])   // stampa il primo elemento dell array
a[0] = 10       // assegnazione
```

La funzione `len` del pkg `builtin` ritorna il # degli elementi dell'array.

Si può usare un array letterale per inizializzare un array con una lista di valori

```
arr := [3]int{1, 2, 3}
var a [3]int = [3]int{1, 2, 3}
```

In un array letterale se al posto della lunghezza si inserisce l'operatore `...`, la size dell'array è determinata dal numero degli elementi inizializzatori (elementi dentro le parentesi graffe).

```
arr := [...]int{1, 2, 3, 4} // arr ha size 4, e tipo [4]int
```

> **NB:** La size di un array deve essere un valore costante

Se un elemento di un array è **comparabile**, allora anche il tipo dell'array è comparabile, quindi possiamo confrontare due arrays di quel tipo usando l'operatore `==`, il quale restituisce `true` se tutti gli elementi di un array sono uguali all'altro

```
arr1 := [4]int{1, 2, 3, 4}
arr2 := [4]int{1, 2, 3, 4}

fmt.Println(arr1 == arr2)   // stampa: true
```

Slices types

Uno slice rappresenta una **sequenza variabile** i cui elementi hanno tutti lo stesso tipo (array dinamici).

Il tipo slice è scritto `[]T` e contiene elementi di tipo `T`. È come la dichiarazione di un array ma senza size.

Array e slice sono connessi. Uno slice è una struttura dati leggera che consente l'accesso ad una sottosequenza di elementi di un array, il quale è chiamato **slice's underlyng array**, cioè l'array sottostante lo slice.

Uno slice ha 3 componenti:

- **puntatore**: punta al primo elemento dell'array che è raggiungibile attraverso lo slice
- **lunghezza**: è il numero degli elementimenti dello slice. Non può eccedere la capacità
- **capacità**: è il numero di elementi compresi tra l'inizio dello slice e la fine dell'array sottostante

Più slice possono condividere lo stesso array sottostante e possono riferirsi a parti sovrapposte di quel array (figura sopra e codice sotto).

Uno slice è formato specificando 2 indici, un limite inferiore ed uno superiore, separati dal''operatore di slice `:` .

```
s := [7]int{1, 2, 3, 4, 5, 6, 7} // dichiarazione array inizializzato con
array letterale

s1 := s[0:3]    // len == 3, capacity == 7, array sottostante == s
s2 := s[3:5]    // len == 2, capacity == 4, array sottostante == s
```

Con l'operatore di slice viene selezionato un intervallo semiaperto che include il primo elemento ma esclude l'ultimo. La seguente espressione crea uno slice che include gli elementi dall'indice 1 al 2.

```
a := [3]string{"stone", "scissor", "paper"} // creazione array
s := a[1:3]
```

È possibile omettere il limite superiore o inferiore (o entrambi).

```
a[:]    // slice contente tutti gli elemnti dell'array a
a[3:]   // slice da elemento di indice 3 di a fino all'ultimo
a[:4]   // slice dal primo elemento di a fino al 4 (indice 3)
```

Fare slicing oltre la capacità dell'array sottostante causa un `panic`, mentre lo slicing oltre la lunghezza dell'array sottostante estende lo slice così che il risultato può essere più grande dell'originale.

```
s := []int{1, 2, 3, 4, 5, 6, 7}

original := s[2:5]

fmt.Println(original[:20])  // genera un panic
newer := original[:5]       // ok
```

> **NB:** Dal momento che uno slice contiene un puntatore all'elemento di un array, passando uno slice ad una funzione è possibile modificare gli elementi dell'array sottostante.

L'espressione che inizializza uno slice è differente da quella che inizializza un array. Uno slice letterale è come un array letterale ma la size è omessa. Questo implicitamente crea una array della size corretta (ossia del # di elementi presenti nelle parentesi graffe) e genera uno slice che punta all'array.

```
s := []int{1, 2, 3, 4, 5, 6, 7} // dichiarazione slice iniziallizzato con
slice letterale
```

Il "valore zero" di uno tipo `slice` è `nil`.

A differenza di un array, uno slice non è comparabile. Dunque non è possibile usare l'operatore `==` per verificare se due slice contengono lo stesso numero di elementi. L'unica comparazione di slice legale è con `nil`.

Un slice con valore `nil` **non ha** un array sottostante , di conseguenza ha *length* e *capacity* uguali a zero, ma ci sono slice `non-nil` (quindi con un array sottostante) che hanno length e capacity uguale a zero come `[]int{}` oppure `make([]int, 3)[3:]`.

La funzione `make` del pkg `builtin` crea uno slice di un tipo specificato di elementi, length e capacity. L'argomento capacity può essere omesso, in tal caso la capacity viene impostata uguale alla length.

```
make([]T, len, cap)
make([]T, len)  // cap == len
```

Uno slice può contenere qualsiasi tipo, anche altri slice.

Per inserire un elemento in uno slice i nGO si usa la funzione `append` del pkg `builtin`.

```
func append(s []T, vs ...T) []T
```

Il primo parametro `s` è uno slice di tipo `T`, e il resto sono 0..n valori di tipo `T` da aggiungere allo slice.

Il risultato della funzione `append` è uno slice contente tutti gli elementi dello slice originale con l'aggiunta dei valori forniti come parametri.

Se l'array sottostante ad s (primo parametro) è troppo piccolo (capacità) per contenere tutti i nuovi valori allora viene allocato un nuovo array. Lo slice ritornato punterà a questo nuovo array.

Maps types

Una hash table è una collezione non ordinata di coppie chiave/valore nella quale tutte le chiave sono distinte e i valori associati ad esse possono essere recuperati, aggiornati o rimossi.

In Go una mapè un riferimento ad un a hash table e un tipo `map` è scritto `map[K]V` dove `K` e `V` sono rispettivamente i tipi di chiave e valore contenuti al suo interno.

Tutte le chiavi in una determinata mappa sono dello stesso tipo, e tutti i valori sono dello stesso tipo ma le chiavi non possono essere dello stesso tipo dei valori.

Il tipo k delle chiavi deve essere un tipo **comparabile** usando l'operatore `==` , in tal modo è possibile verificare se una data chiave è uguale in un già presente.

La funzione `make` del pkg `builtin` può essere usata per creare una `map`.

```
m := make(map[K]V)
```

È inoltre possibile usare una map letterale per creare una nuova `map` popolata con delle coppie chiave/valore.

```
m := map[string]int{
    "samuele": 27,
    "luca":    26,
}

// è equivalente a
m := make(map[string]int)
m["samuele"] = 27
m["luca"] = 26

// un alternativa per creare un nuova mappa vuota
m := map[string]int{}
```

Gli elementi di un `map` sono **accessibili** attraverso l'usuale notazione di subscripting

```
m["samuele"] = 29   // modifica l'elemento m["samuele"]
```

Gli elementi di una mappa possono essere **rimossi** con la funzione `delete` del pkg `builtin` .

```
delete(m,"samuele") // rimuove l'elemento completo m["samuele"], sia chiave che valore
```

Tutte queste operazione sono sicure anche se gli elementi non sono presenti nella `map` . accedere ad una `map` usando una chiave che non è presente ritorna il "valore zero" del tipo dei valori della mappa.

> **NB**: Quando si accede ad un elemento della mappa tramite subscripting troviamo sempre un valore. Se la chiave è presente, verrà restituito il corrispondente valore, altrimenti veeà restituito il valore zero del tipo dei valori. Spesso vogliamo sapere se l'elemento con una certa chiave è realmente presetne ed ha chiave con valore 0, oppure se non è presente nella `map` . Per fare ciò si usa il cosidetto costrutto **comma ok**.

```
value, ok := m["samuele"]
if !ok {...}    // valore con chiave "samuele" non presente nella map m

// forma equivalente
if value, ok := m["samuele"]; !ok {...}
```

Fare lo subscripting di una `map` produce quindi 2 valori:

- il primo il valore di `m["samuele"]`
- il secondo è un booleano che ci dice se esiste l'elemento con quella chiave o no

Come per le `slice` , una `map` non può essere confrontata con un'altra `map` . L'unico confronto possibile è con il valore `nil` . Il "valore zero" di uno tipo `map` è `nil` .

Structs types

Una `struct` è un tipo di dato aggregato che raggruppa 0 o più valori di qualsiasi tipo vedendoli come singole entità. Ogni valore è chiamato field (campo).

```
type Person struct {
    field1 string
    field2 int
}
```

I campi di una `struct` sono accessibili attraverso l'operatore `.` dot. I campi di una `struct` possono anche essere acceduti attraverso un puntatore alla `struct` . Per fare ciò si dovrebbe prima dereferenziare il puntatore e poi accedere al campo della struttura `(*p).X` . Tuttavia il linguaggio permette di scrivere direttamente `p.X` senza dereferenziare esplicitamente.

Un `struct` di tipo `S` non può contenere un valore di tipo `S`, ma può dichiarare un field puntatore a `S`.

Il "valore zero" di una `struct` è composto dal valore zero di ognuno dei suoi fields.

I fields di solito sono scritti uno per riga, con il nome che precede il tipo (vedi sopra). Campi consecutivi con lo stesso tipo possono essere combinati.

```
type Person struct {
    field1, field2 string
    field2 int
}
```

> **NB**: il nome di un field di una `struc` è "exported" se comincia con la lettera maiuscola.

Il tipo `struct` senza fields è chiamato *empty struct* e si dichiara con `struct{}`. Ha size pari a zero.

```
var s struct {}

// forma equivalente

s := struct{}{}
```

Struct Literals

Una `struct` letterale rappresetna una nuova struttura allocata, nella quale vengono elencati i valori dei suoi campi. Ci sono 2 forme per una struttura letterale

1. La prima richiede che i valori siano specificati per *ogni* campo, nel giusto ordine.

```
p := Person{"Samuele", 27}
```

2. Nella seconda un valore della `struct` è inizializzato elencando alcuni o tutti i campi della struttura e i suoi corrispondenti valori.

```
// p ha tipo Person
p := Person {
    name: "Samuele",
    age:  27,
}
```

L'operatore prefisso `&` associato ad una struttura letterale ritorna l'indirizzo di tale `struct`.

```
// p ha tipo *Person
p := &Person {
    name: "Samuele",
    age:  27,
}
```

Se tutti i campi di una `struct` sono confrontabili, la `struct` stessa è confrontabile. Di conseguenza due `struct` dello stesso tipo possono essere comparate usando gli operatori `==` e `!=` . L'operatore `==` confronta i fields corrispondenti delle due strutture in ordine.

In Go esiste un meccanismo chiamato **struct embedding** che consente di usare il nome di un tipo `struct` come *anonymous filed* di un altro tipo `struct`, fornendo un sintassi abbreviata che semplifica la "dot expression" per accedere ai campi della `struct`.

```
type Circle struct {
    X, Y, Radius int
}

type Wheel struct {
    X, Y, Radius, Spokes
}

// fattorizziamo usando struct embedding

type Point struct {
    X, Y int
}

type Circle struct {
    Center Point
    Radius int
}

type Wheel struct {
    Circle Circle
    Spokes int
}
```

Se alla `struct` integrata è stato assegnato un nome, per accedere ai sui field bisognerà usare la "dot expression" fornendo il "path" completo, come nell'esempio

```
var w Wheel
w.Circle.Center.X = 8
```

Se invece usiamo un field anonimo è possibile accedere direttamente ai fields delle `struct` integrata.

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

type Circle struct {
    Point   // field anonimo
    Radius int
}

type Wheel struct {
    Circle  // field anonimo
    Spokes int
}

var w Wheel
w.X = 8 // accedo direttamente al field X di Point
```

Poichè i *fields anonimi* hanno nomi impliciti, non è possibile avere due fields anonimi dello stesso tipo, in quanto si incorre in un conflitto di nomi. Per ovviare il problema basta assegnare un nome alla `struct` integrata dovendo però fornire il "path completo" per accedere ai campi della `struct` integrata.

Per costruire una `struct` letterale contenente che contiene una `struct` integrata si usa la seguente sintassi.

```
type Person struct {
    name string
    age  int
    Address // anonymous embedded struct
}

type Address struct {
    street string
    number int
}

// sintassi 1) esempio sopra
p1 := Person{"Samuele", 27, Address{"Viale Trieste", 429}}

// sintassi 2) esempio sopra
p2 := Person{
    name: "Samuele",
    age:  27,
    Address: Address{
        street: "Viale Trieste",
        number: 429,
    },
```

```
}
```

In questo caso la `struct Address` assume lo stesso nome del tipo.

Le funzioni new e make

Go ha due primitive di allocazione, le funzioni `make` e `new` del pk `builtin`. La funzione `new(Type) *Type` alloca memoria ma non la inizializza, la "mette" al valore zero del tipo `Type`. Nella terminologia di Go, si dice che la funzione `new(T)` ritorna un puntatore al "valore zero" appena allocato di tipo `T` .

La funzione `new` si può utilizzare per tutti i tipi tranne che per i tipi reference come `map`, `slice` e `channel` per i quali si usa la funzione `make`.

La funzione `make(T, args` crea `map`, `slice` e `channel` e ritorna un valore inizializzato (non un valore zero) di tipo `T`. La ragione della distinzione rispetto alla `new` è dovuta al fatto che questi tipi in realtà rappresentano dei riferimenti a strutture dati che devono essere inizializzate prima dell'uso.

Functions types

Una dichiarazione di funzione in GO ha un *nome*, una *lista di parametri*, una *lista opzionale di risultati* ed un *corpo*.

```
func name(parameter-list) (result-list) {
    body
}
```

La lista dei parametri specifica i nomi e i tipi dei parametri della funzione. **Tutti i parametri di una funzione sono passati per valore**. La lista dei risultati specifica il tipo (volendo anche il nome) dei valori che la funzione ritorna. Una funzione può ritornare qualsiasi numero di risultati (1,2,...n). Se la funzione ritorna un risultato senza nome (solo tipo) o nessun risultato, le parentesi tonde possono essere omesse.

La lista dei parametri può avere 0..n parametri.

```
func f() int {...}
func f(i int) int {...}
func f(i int, s string) {...}
```

Quando 2 o piu parametri di una funzione condividono un tipo, è possibile omettere il tipo ed inserirlo solo in fondo alla dichiarazione.

```
func f(i, y int) int {...}
```

Si può dare un nome ai valori di ritorno. Se si decide di dare un nome alle variabli da restiture dalla funzione la sintassi è la seguente:

```go
func f(i int) (x, y int) {
    x = 5 + 4
    y = x * 4
    return
}
```

I nomi dati alle variabili di ritorno dovrebbero essere significativi per documentare il tipo di ritorno.

Lo statement `return` senza gli argomenti viene detto **naked** e ritorna i/il valori/e di ritorno a cui è stato dato un nome.

Valori Funzione

Le funzioni in Go sono *first-class value*: come altri valori **le funzioni hanno dei tipi** e possono essere **assegnate** ad una variabile o **passate** ad una funzione o **ritornate** da una funzione.

Il valore di un variabile di tipo `func` non inizializzata è `nil` (cioè il valore zero di una funzione).

```go
var f func() int    // funzione dichiarata ma non inizializzata, ha valore nil

fmt.Printf("%T\n", f)   // stampa il tipo: func() int
fmt.Println(f == nil)   // stampa: true

// dichiarazione fuori dal main (ovviamente)
func f1() int { ... }   // f1 ha tipo func() int
func f2(n int) string { ... } // f2 ha tipo func(int) string

func f3() func() int { // ritorna una funzione
    return f1
}
```

Un valore funzione può essere chiamato come qualsiasi altra funzione.

```go
x := 4
func fun(x int) int {
    return x * x
}
f := fun    // valore funzione asseganto a variabile
fmt.Println(f())    // stampa: 16
```

I valori funzione possono essere confrontati solo con il valore `nil`.

Funzioni anonime

Le funzioni con nome possono essere dichiarate solo a livello di package, ma è possibile usare una *funzione letterale* per denotare un valore funzione con una espressione. Una funzione letterale è scritta come una dichiarazione di funzione ma senza un nome che segue la keyword `func`. Si tratta di un'espressione e il suo valore è chiamato *funzione anonima*.

```
f := func() string {
    return "hello"
}

fmt.Println(f())    // stampa : hello

// funziona anonima dichiarata e chiamata
func() int {
    return 1
}()
```

Funzioni Closures

Le funzioni in Go possono essere *closures*. Una closure è un valore funzione che fa riferimento a variabili esterne al suo corpo. Questa funzione può accedere e assegnare valori alla variabili a cui fa riferiemento.

```
func adder() func(int) int {
    sum := 0
    return  func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos := adder()
    for i := 0; i < 10; i++ {
        fmt.Print(pos(i), " ")
    }
}

// stampa: 0 1 3 6 10 15 21 28 36 45
```

Funzioni Variadic

Una *funzione variadic* può essere chiamata con un numero variabile di argomenti. Per dichiarare una *funzione variadic* il tipo del parametro finale della funzione deve essere preceduto dal''operatore `...`, il quale indica che la funzione può essere chiamata con un qualsiasi numero di argomenti di quel tipo

```
func f(s string, i ...int) { ... }
```

Implicatamente Il chiamante (chi chiama la funzione) alloca un array, copia i valori al suo interno e passa uno slice dell'intero array alla funzione.

Anche se `...int` si comporta come uno slice all'interno del corpo della funzione, il tipo di una *funzione variadic* è diverso dal tipo di una funzione con un parametro di tipo slice.

```
func f(s string, i ...int) { ... }
func s(s string, sli []int) { ... }

fmt.Printl("%T\n", f)   // stampa func(string, ...int)
fmt.Printl("%T\n", f)   // stampa func(string, []int)
```

Se invece si vuole passare ad una funzione variadic uno slice è possibile farlo con la seguente sintassi

```
func f(s string, i ...int) { ... }
values := []int{1, 2, 3, 4}

fmt.Println(f(values...))
```

Metodi

Un metodo è dichiarato con una variante rispetto alla ordinaria dichiarazione di funzione nel quale compare un parametro extra racchiuso tra parentesi tonde prima del nome della funzione.

Il parametro extra è chiamato *receiver*. Il parametro receiver collega la funzione al tipo di quel parametro.

```
func (receiver) name(parameter-list) (result-list) {
    body
}
```

Per invocare un metodo agganciato ad un tipo si una il *selettore*, il quale selezione il metodo del tipo.

```
p := person {"samuele", 27}
p.SayHello()    // selettore
```

In Go è possibile associare metodi ad ogni tipo, tipi base compreso

Il receiver di un metodo può essere anche di tipo puntatore. Questo significa che i metodi con receiver di tipo puntatore possono modificare il valore a cui puntano. Cosa non possibile dichiarando un receiver come tipo non puntatore.

```go
func (p *person) setName(name string) {
    p.name = name   // side effect su p
}

func (p person) setname(name string) {
    p.name = name   // nessun side effect, p è una copia
}
```

Funzioni con un argometno di tipo puntatore **devono** ricevere un puntatore come tipo. Invece metodi con *receiver* di tipo puntatore possono essere invocati su sia su un valore che su un puntatore.

```go
func (p *person) setName(name string) {
    p.name = name
}

func (p person) setAge(age int) {
    p.age = age
}

p := &person{"Samuele", 27}
p2 := *p

p.setName("Marco")  // ok
p.setAge(45)        // ok
p2.setName("Luca")  // ok
p2.setAge(33)       // ok
```

È consigliato usare *receiver* di tipo puntatore:

- è possibile fare side effetcs
- si evita la copia del valore ad ogni chiamata al metodo

Method sets

Un insieme di metodi determina quali metodi sono associati ad un tipo.

L'insieme dei metodi di un *tipo interfaccia* **è** la sua interfaccia.

L'insieme dei metodi di qualsiasi altro tipo `T` invece è composto da tutti i metodi dichiarati con *receiver* di tipo `T`.

L'insieme dei metodi del corrispondente tipo puntatore `*T` è l'insieme di tutti i metodi dichiarati con *receiver* di tipo `*T` oppure `T`, ossia contiene anche l'insieme dei metodi di `T`.

L'insieme di metodi di un tipo determina le interfacce che il tipo implementa e i metodi che possono essere chiamati utilizzando un ricevitore di quel tipo.

Quando un tipo `T` implementa un'interfaccia `I` (implementando i suoi metodi) assume anche il tipo `I`. Di conseguenza è possibile usare un `T` quando è richiesto un tipo `T` o un tipo `I`.

Valgono le seguenti regole:

- Tuttavia quando `T` implementa un metodo `m` dell'interfaccia `I`, se il *receiver* di `m` ha tipo `T`, il metodo viene aggiunto al **method set** di `T` e di `*T`;
- Se inece il *receiver* di `m` ha tipo `*T`, il metodo viene aggiunto solo al **method set** di `*T`

```
/* Receivers        Values */
(t T)               T and *T
(t *T)              *T
```

Di conseguenza qualora si dichiari un *receiver* di tipo puntatore `*T` per utilizzare una valore `T` al posto di un'interfaccia da lui implementata bisognerà fornire il tipo `*T`, ossia un puntatore a `T`.

```
type person struct {
    name string
    age  int
}

type speaker interface {
    speak()
}

func (p person) getName() string {
    return p.name
}

func (p *person) speak() {
    fmt.Println("hello", p.getName(), "i'm", p.getAge())
}

func (p *person) getAge() int {
    return p.age
}

func main() {
    p := person{"sam", 23}
    spk(p)  // errore: è richiesto uno tipo *p
    spk(&p) // ok
}

func spk(s speaker) {
    s.speak()
}
```

Interfacce

Un'interfaccia è un **tipo** astratto ed è definito come un insieme di signature di metodi.

Un valore di tipo interfaccia può contenere qualsiasi valore che implementi i sui metodi.

```
type Speaker interface {
    speak(message string) string
}
```

Un tipo implementa un'interfaccia implementando i suoi metodi. Non ci sono dichiarazioni esplicite per implementare un interfaccia. Viene fatto automaticamente dal linguaggio. Questo rende possibile adattare tipi a codice già scritto al quale non è possibile mettere mano.

La regola di assegnazione per le interfacce è molto semplice: un espressione può essere assegnata ad un interfaccia solo se il suo tipo soddisfa quell'interfaccia, ossia implementa tutti i suoi metodi.

Un interfaccia senza metodi è chiamata tipo *interfaccia vuota* (**empty interface**) e poichè non richiede alcun requisito per essere soddisfatta (nessun metodo da implementare), è possibile assegnare qualsiasi valore ad una interfaccia vuota.

Il "valore zero" di un interfaccia è `nil`

```
type any interface{}    // dichiarazione nuovo tipo con tipo sottostante
interfaccia vuota

func main() {
    var a any   // dichiarazione variabile di tipo any (alias di
interfaccia vuota)
    fmt.Printf("tipo: %T, valore: %v\n", a, a)  // stampa: tipo: <nil>,
valore: <nil>

    var a2 interface{} // altra dichiarazione interfaccia vuota
    fmt.Printf("tipo: %T, valore: %v\n", a2, a2)    // stampa: tipo: <nil>,
valore: <nil>

    a2 = true
    fmt.Printf("tipo: %T, valore: %v\n", a2, a2)    // stampa: tipo: bool,
valore: true

    a2 = 12
    fmt.Printf("tipo: %T, valore: %v\n", a2, a2)    // stampa: tipo: int,
valore: 12

    a2 = "hello"
    fmt.Printf("tipo: %T, valore: %v\n", a2, a2)    //stampa: tipo: string,
valore: hello
}
```

Valori interfaccia

Concettualmente un valore di un tipo interfaccia è composto da due componenti (una tupla): un tipo concreto e un valore di quel tipo.

```
(value, type)
```

Un valore interfaccia contiene un valore del suo tipo concreto sottostante.

Chiamare un metodo su un valore interfaccia esegue il metodo con lo stesso nome del suo tipo sottostante.

Se il valore concreto all'interno dell'interfaccia stessa è pari a zero, il metodo verrà chiamato con un *receiver* `nil`. Ciò non produce un errore (tipico null pointer exception in altri linguaggi).

Si noti che un valore interfaccia che contiene un valore concreto `nil` è essa stessa non-nulla.

Un valore nullo di tipo interfaccia non ha nè valore nè tipo. Chiamare un metodo su di un'interfaccia `nil` produce un errore a run time in quanto non è presente alcun tipo all'interno della tupla dell'interfaccia per indicare quale metodo concreto chiamare

Type Assertions

Un' asserzione di tipo fornisce l'accesso al valore concreto sottostante il valore dell'interfaccia. È una sort di type checking.

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

Questo statement asserisce che il valore `i` dell'interfaccia contiene il tipo concreto `T` e assegna il valore `T` sottostante alla variabile `t` .

Se `i` non contiene un valore di tipo `T`, lo statement lancerà un *panic* (un error a run time).

Per *verificare* se il valore di un'interfaccia contiene un tipo specifico, un *type assertion* può ritornare 2 valori:

- il tipo sottostante
- un valore booleano che indica se il valore di quel tipo è presente o meno

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

Se i contiene un `T` , allora `t` sarà il valore sottostante e `ok` sarà `true`. Altrimenti, `ok` sarà falso e `t` sarà `0` e non occorrerà alcun *panic*. (nota la somiglianza con lo statement *comm ok* delle `map`).