PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

GO

Go è un linguaggio di programmazione open source multipiattaforma, può essere utilizzato per creare applicazioni ad alte prestazioni, è un linguaggio compilato veloce, tipizzato staticamente che sembra un linguaggio interpretato e digitato dinamicamente, ha la sintassi simile a C++. Go ha tempi di esecuzione e tempi di compilazione rapidi, supporta la concorrenza e ha la gestione della memoria.

Ogni programma in Go ha una dichiarazioni dei package, significa che il programma appartiene al package "main".
L'import dei package, in questo caso abbiamo importato i file inclusi nel package "fmt" utili per input e output.
NOTA: La parentesi aperta "{" non può trovarsi a inizio riga, altrimenti è un errore di sintassi.

```
package main

import "fmt"

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

Per eseguire il programma, metti il codice in hello-world.go e invoca go run in un terminale.

```
$ go run hello-world.go
hello world
```

A volte ci sarà la necessità di compilare i nostri programmi in un file binario. Possiamo fare ciò usando go build. Dopodiché, possiamo eseguire il binario compilato direttamente con `./hello-world`

```
$ go build hello-world.go
$ ls
hello-world     hello-world.go
```

Go ha tre ***tipi di dati*** di base:

- ***Booleani*** : rappresenta un valore booleano ed è vero o falso (Sintassi: bool)
- ***Numerico*** : rappresenta tipi interi, valori in virgola mobile e tipi complessi (Sintassi: int / float32 / float64)
- ***Stringa*** : rappresenta un valore stringa (Sintassi: string)

Go supporta tutti gli ***operatori*** classici, come operatori aritmetici (+, -, *, /, %, ++, --), operatori di assegnazione (=, +=, -=, ecc…), operatori di confronto (<, >, ==, !=, <=, >=), operatori logici (&&, ||, !) e operatori bit a bit (&, |, ^, <<, >>).

VARIABILI:

In Go, le ***variabili*** sono dichiarate esplicitamente e sono usate dal compilatore. Per dichiarare una variabile si usa la sintassi:

var varName type = value (bisogna SEMPRE specificare *type* OPPURE *value*)

La parola chiave *var* dichiara una o più variabili.

È possibile dichiarare più variabili sulla stessa riga.

Se non specificato il *type*, Go dedurrà il tipo delle variabili inizializzate.

Le variabili dichiarate senza inizializzazione corrispondono l loro cero-valued, ogni tipo ha il suo, come ad esempio int è 0 e string è "".

La sintassi := è una abbreviazione per dichiarare e inizializzare una variabile, in questo caso è l'abbreviazione di var f string = "short".

NOTA: Con la parola chiave ":=" il tipo della variabile viene dedotto dal valore, le variabili così dichiarate sono visibili solo all'interno delle funzioni e il loro valore va assegnato sulla stessa riga.

```
func main() {
    var a string = "init"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)

    f := "short"
    fmt.Println(f)
}
```

```
init
1 2
true
0
short
```

COSTANTI:

Go ammette l'utilizzo di ***costanti*** di tipo string, boolean, e di tipo numerico. Si utilizza la seguente sintassi:

const CONSTNAME type = value (*value* di una costante deve essere assegnato quando lo si dichiara).

NOTA: Le costanti sono immodificabili e di sola lettura, e sono scritte in MAIUSCOLO.
La costante STRINGA è tipizzata in quanto specificato il tipo.

La keyword const può essere utilizzata ovunque la keyword var è ammessa.

Le espressioni costanti vengono calcolate in aritmetica a precisione arbitraria.

Una costante numerica non ha un tipo fin quando non gli viene assegnato esplicitamente, ad esempio tramite un cast.

```
const STRINGA string = "costante"

func main() {
    fmt.Println(STRINGA)

    const n = 500000000

    const d = 3e20 / n
    fmt.Println(d)

    fmt.Println(int64(d))
}
```

```
costante
6e+11
600000000000
```

IF/ELSE:

Modificare il flusso di controllo con ***if*** ed ***else*** in Go è semplice e ricalca la classica sintassi vista in altri linguaggi.

È possibile avere un comando if senza il ramo else.

Un comando può precedere il test del comando if. Qualsiasi variabile dichiarata in questo comando è visibile all'interno di tutti i rami del comando if

NOTA: non sono necessarie le parentesi intorno alle condizioni del comando if in Go, ma le parentesi graffe sono necessarie.
NOTA: Non esiste un operatore condizionale ternario (forma rapida con ?) in Go. Quindi è necessario utilizzare un comando if completo anche per semplici istruzioni/test.

```
func main() {
    if num := 9; num < 0 {
        fmt.Println(num, "è negativo")
    } else if num < 10 {
        fmt.Println(num, "ha una cifra")
    } else {
        fmt.Println(num, "ha più di una cifra")
    }
}
```

```
9 ha una cifra
```

SWITCH:

Gli ***switch*** esprimono condizionali attraverso più rami.

Puoi utilizzare le virgole per dividere più espressioni nella stessa dichiarazione case. In questo esempio utilizziamo anche il caso opzionale default, che viene eseguito nel caso l'espressione non possa essere valutata in nessuno dei rami precedenti.

Uno switch senza espressione (es senza day) è un metodo alternativo per esprimere la logica degli if/else.

```
func main() {
    day := 8
    switch day {
    case 1:
        fmt.Println("Inizio settimana")
    case 2, 3, 4, 5:
        fmt.Println("Settiama")
    case 6, 7:
        fmt.Println("Fine settimana")
    default:
        fmt.Println("Non è un giorno della settimana")
    }
}
```

CICLO FOR:

Il ***for*** è l'unico costrutto per eseguire cicli in Go.

Un classico ciclo for inizializzazione/test/incremento.

La parola chiave *continue* farà saltare una iterazione a condizione verificata, mentre *break* interrompe il ciclo di iterazione.

```
func main() {
    for i := 0; i < 5; i++ {
        if i == 1 {
            continue
        }
        if i == 4 {
            break
        }
        fmt.Println(i)
    }
}
```

```
0
2
3
```

ARRAY:

In Go, un ***array*** è una sequenza numerata di elementi con una lunghezza specifica. La sintassi è la seguente:

var array_name = [length]datatype{values} OPPURE *var array_name = [...]datatype{values}*

Qui creeremo un array chiamato a che conterrà esattamente 5 elementi di tipo int. Il tipo degli elementi e la lunghezza sono entrambi parti integranti del tipo dell'array. Per default gli array vengono inizializzati allo zero value, che per gli elementi di tipo int significa essere inizializzati a 0.

Possiamo assegnare un valore ad uno specifico indice utilizzando la classica sintassi: array[indice] = valore , e possiamo ottenere il valore con array[indice].

La funzione builtin *len* restituisce la lunghezza dell'array.

Anche con gli array è possibile utilizzare ":=". È possibile dichiarare e inizializzare un array nella stessa linea.

NOTA: Se la dimensione non viene specificata, dichiarando l'array con […], essa viene dedotta dai valori di inizializzazione (ricordando che un array ha dimensione fissata).

```
func main() {
    var a [5]int
    fmt.Println("emp:", a)

    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    fmt.Println("len:", len(a))

    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)
}
```

```
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
```

SLICE:

Gli ***Slice*** sono un data type fondamentale di Go, e rendono la gestione degli array più semplice e potente. A differenza degli array, gli slice vengono definiti dando soltanto il tipo degli elementi che contengono (non il numero di elementi). La sintassi è la seguente:

slice_name := []datatype{values} OPPURE *slice_name := make([]type, length, capacity)*, *capacity* può essere omessa

Per creare uno slice vuoto con una lunghezza diversa da 0, usa la funzione make. Di seguito creiamo uno slice di string di lunghezza 3 (all'inizio zero-valued).

Possiamo impostare e prendere valori esattamente come negli array.

len restituisce, come ci si potrebbe aspettare, la lunghezza dello slice.

Oltre a queste operazioni di base, gli slice ne hanno molte altre che permettono loro di essere più funzionali degli array. Una di queste è la funzione append, che restituisce uno slice contentente uno o più ulteriori valori.

Gli slice possono anche essere copiati con la funzione copy. Di seguito creiamo uno slice vuoto c della stessa lunghezza di s e copiamo i valori di s in c.

Gli slice supportano un operatore "slice" che ha la sintassi *variabileSlice[inizio:fine]* con *inizio* compreso e *fine* non compreso. Per esempio, qui generiamo uno slice con gli elementi s[2], s[3] e s[4].

NOTA: Fare questa operazione non comporta una copia dei dati ma un suo riferimento, se viene modificata questo "slice" anche l'originale viene modificato.

Possiamo, inoltre, dichiarare e inizializzare uno slice in una sola riga.

```
func main() {
    s := make([]string, 3)
    fmt.Println("emp:", s)

    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    fmt.Println("len:", len(s))

    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)

    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    l := s[2:5]
    fmt.Println("sl1:", l)

    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)
}
```

È possibile anche unire più slice con append, ad esempio:
slice3 = append(slice1, slice2...), i "..." sono necessari quando si uniscono più slice

NOTA: la funzione *cap(nomeSlice)* restituisce la capacità (il numero di elementi a cui può crescere o ridursi)

```
emp: [  ]
set: [a b c]
get: c
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
dcl: [g h i]
```

MAP:

Le ***Map*** sono la struttura built-in di Go per gli Array associativi (in altri linguaggi si possono trovare strutture simili sotto il nome di hash table o dizionari). Una mappa è una raccolta non ordinata e modificabile che non consente duplicati. Ogni elemento in una mappa è una coppia chiave:valore.
La sintassi è la seguente:

var a = map[KeyType]ValueType{key1:value1, ...} OPPURE *a := map[KeyType]ValueType{key1:value1, ...}*

Per creare una mappa vuota:

var a = make(map[KeyType]ValueType) OPPURE *b := make(map[KeyType]ValueType)*

NOTA: la funzione make() è il modo giusto per creare una mappa vuota. Se crei una mappa vuota in un modo diverso e ci scrivi, causerà un panico durante il runtime.

Per creare una nuova map vuota, utilizza la funzione built-in make.

Puoi impostare i valori della map utilizzando la sintassi tipica nomemap[chiave] = valore.

Passare la map ad una funzione di stampa (tipo Println) mostrerà tutte le coppie chiave-valore della map.
Puoi ottenere il valore di una chiave con nomemap[chiave].

La funzione built-in len restituisce il numero di coppie chiave-valore se la si invoca su una map.
La funzione built-in delete rimuove le coppie chiave-valore dalla map.

Quando si accede ad una map è possibile controllare il secondo valore restituito opzionale che indica la presenza o meno di una chiave all'interno di una map. Questo parametro può essere utilizzato per discernere il caso in cui una chiave non è presente dal caso in cui una chiave ha assegnato lo zero-value (ad esempio 0 o ""). In questo caso non abbiamo nemmeno bisogno del valore della chiave, per cui scartiamo il primo parametro utilizzando l'identificatore blank "_"

È anche possibile dichiarare e inizializzare una nuova map con la sintassi dove si inizializza.

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

    m["k1"] = 7
    m["k2"] = 13

    fmt.Println("map:", m)

    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    fmt.Println("len:", len(m))

    delete(m, "k2")
    fmt.Println("map:", m)

    _, prs := m["k2"]
    fmt.Println("prs:", prs)

    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)
}
```

```
map: map[k1:7 k2:13]
v1:  7
len: 2
map: map[k1:7]
prs: false
map: map[bar:2 foo:1]
```

RANGE:

I ***range*** statement permettono di iterare sugli elementi di una varietà di strutture di dati, similmente ai ***foreach*** di altri linguaggi. La sintassi:

for index, value := range array|slice|map { }

Di seguito utilizziamo un range per sommare i numeri di uno slice. (Con gli array viene fatto allo stesso modo).

range sugli array e sugli slice restituisce sia l'indice sia il valore di ognuno degli elementi. Prima non avevamo bisogno di utilizzare l'indice, quindi l'avevamo ignorato utilizzando il blank _. Qualche volta potremmo avere anche solo bisogno dell'indice, e ignorare il suo valore.

Utilizzare range su una map itera sulle coppie chiave-valore.

```go
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("somma:", sum)

    for i, num := range nums {
        if num == 3 {
            fmt.Println("indice:", i)
        }
    }

    kvs := map[string]string{"a": "alice", "b": "bob"}
    for k, v := range kvs {
        fmt.Println(k, " -> ", v)
    }
}
```

```
somma: 9
indice: 1
b  ->  bob
a  ->  alice
```

FUNZIONI:

Una ***funzione*** è un blocco di istruzioni che può essere utilizzato ripetutamente in un programma. Si utilizza la seguente sintassi:

func functionName(param1 type, param2 type, param3 type) type {},

NOTA: L'ultimo type è il valore di ritorno, questo può essere omesso se non c'è ritorno

Go supporta ***valori restituiti multipli***, similmente a python. Questa funzionalità è usata spesso nel Go idiomatico, per esempio per restituire sia il valore sia l'eventuale errore nell'esecuzione di una funzione.

È possibile dichiarare la variabile di ritorno nella firma della funzione. La variabile sol è già dichiarata, si omettono i ":". È possibile scrivere solo return senza specificare a quale variabile si riferisce in quanto già esposta nella firma

Le funzioni in Go possono restituire anche più valori di ritorno.
Nelle funzioni con parametri multipli dello stesso tipo si può omettere il tipo per i parametri consecutivi che hanno lo stesso tipo, e indicare il tipo solo per l'ultimo parametro.

Utilizzando il *multiple assignment* creiamo due diverse variabili dai valori restituiti della funzione molt_div().

```go
func somma(x int, y int) int {
    sol := x + y
    return sol
}
func sottrazione(x int, y int) (sol int) {
    sol = x - y
    return
}
func stampa(operazione string, sol int) {
    fmt.Println(operazione, sol)
}
func molt_div(x, y int) (molt, div int) {
    molt = x * y
    div = x / y
    return
}
func fattoriale(n int) (fatt int) {
    if n <= 1 {
        fatt = 1
    } else {
        fatt = n * fattoriale(n-1)
    }
    return
}

func main() {
    sol := somma(3, 7)
    stampa("Somma:", sol)
    sol = sottrazione(4, 3)
    stampa("Sottrazione:", sol)
    molt, div := molt_div(10, 5)
    stampa("Moltiplicazione: ", molt)
    stampa("Divisione: ", div)
    fatt := fattoriale(6)
    stampa("Fattoriale: ", fatt)
}
```

```
Somma: 10
Sottrazione: 1
Moltiplicazione:  50
Divisione:  2
Fattoriale:  720
```

FUNZIONI VARIADICHE:

Le ***funzioni variadiche*** possono essere chiamate con un numero arbitrario di parametri in coda. fmt.Println è il classico esempio di una funzione variadica.

Questa è un esempio di funzione che accetta un numero arbitrario di parametri di tipo int.

Le funzioni variadiche possono essere invocate nel classico modo indicando ogni parametro separatamente.

Se i parametri si trovano dentro uno slice, puoi passarlo direttamente ad una funzione variadica tramite la sintassi seguente: nomefunzione(slice...)

```go
func somma(numeri ...int) {
    sol := 0
    for _, num := range numeri {
        sol += num
    }
    fmt.Println(sol)
}

func main() {
    somma(1, 2)
    somma(1, 2, 3)

    numeri := []int{1, 2, 3, 4}
    somma(numeri...)
}
```

```
3
6
10
```

CHIUSURA (FUNZIONI ANONIME):

Go supporta le ***funzioni anonime***, che possono formare delle ***chiusure***. Le funzioni anonime sono utili quando vuoi definire una funzione senza darle un nome.

Questa funzione intSeq restituisce un'altra funzione, che definiamo anonimamente dentro il corpo della funzione intSeq. La funzione restituita racchiude la variabile i per formare una chiusura.

Facciamo una chiamata ad intSeq, assegnando il risultato (una funzione) a nextInt. Il valore di questa funzione racchiude in sé stessa il valore di i, il quale verrà aggiornato la prossima volta che utilizziamo nextInt.

Osserviamo l'effetto della chiusura facendo una chiamata a nextInt un po' di volte.

Per confermare che lo stato è unico a quella funzione particolare, creiamo e testiamone una nuova newInts.

```go
func intSeq() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}

func main() {
    nextInt := intSeq()

    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    newInts := intSeq()
    fmt.Println(newInts())
}
```

```
1
2
3
1
```

RICORSIONE:

Go supporta le ***Funzioni ricorsive***. Ecco il classico esempio di una funzione ricorsiva, il fattoriale.

Questa funzione fact invoca sé stessa fin quando non raggiunge il caso base per n uguale a 0.

```go
func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}
func main() {
    fmt.Println(fact(7))
}
```

```
5040
```

PUNTATORI:

Go permette l'utilizzo dei ***puntatori***, che si traduce nell'abilità di passare riferimenti a valori all'interno del programma.

Dimostreremo come i puntatori funzionino diversamente dai valori tramite 2 funzioni:

- *zeroval* ha un parametro di tipo int, quindi il parametro passato sarà un valore, non un puntatore. Quando chiameremo la funzione zeroval, il suo parametro ival verrà copiato da quello della funzione chiamante;
- *zeroptr* invece ha un parametro di tipo *int, e ciò significa che è un puntatore a un int. L'istruzione *iptr nel corpo della funzione permette di dereferenziare l'indirizzo di memoria puntato da iptr in modo da ottenere il suo valore. Se si assegna un valore ad *iptr si va a modificare il valore all'indirizzo di memoria puntato.

La formula &i restituisce l'indirizzo nella memoria di i, ovvero un puntatore ad i.

Anche i puntatori possono essere stampati.

zeroval non cambia il valore di i in main, zeroptr invece sì perché ha un riferimento al valore nella memoria di quell'indirizzo.

```go
func zeroval(ival int) {
    ival = 0
}
func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("iniziale: ", i)

    zeroval(i)
    fmt.Println("zeroval:  ", i)

    zeroptr(&i)
    fmt.Println("zeroptr:  ", i)

    fmt.Println("puntatore:", &i)
}
```

```
iniziale:  1
zeroval:   1
zeroptr:   0
puntatore: 0xc000012088
```

STRUCT:

In Go le ***struct*** sono collezioni di field (campi) a cui è associato un tipo. Sono utili per raccogliere insieme dati in modo da formare dei record. Sintassi:

type struct_name struct { member1 datatype; ...; }

Questa struct person possiede due campi, rispettivamente name ed age.

Con questa sintassi si crea una nuova struct.

Puoi indicare il nome del campo quando crei una struct.

I field non indicati verranno inizializzati con il loro zero-value.

Inserire un & a prefisso della dichiarazione permetterà di ottenere un puntatore alla struct.

Puoi accedere ai campi della struct con l'operatore "." (punto).

Puoi utilizzare il punto anche per i puntatori a struct. Il puntatore verrà dereferenziato automaticamente.

Le struct sono mutabili.

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

func main() {
    fmt.Println(person{"Nicola", 20})

    fmt.Println(person{name: "Luigi", age: 30})

    fmt.Println(person{name: "Alessandro"})

    fmt.Println(&person{name: "Luca", age: 40})

    s := person{name: "Mario", age: 50}
    fmt.Println(s.name)

    sp := &s
    fmt.Println(sp.age)

    sp.age = 51
    fmt.Println(sp.age)
}
```

```
{Nicola 20}
{Luigi 30}
{Alessandro 0}
&{Luca 40}
Mario
50
51
```

METODI:

Go permette l'utilizzo dei ***metodi*** definiti su tipi all'interno di uno stesso package.

Questo metodo area ha un ***receiver type*** *rect.

I metodi possono essere definiti o per un puntatore o per semplicemente un valore. Ecco un esempio di metodo *perim()* con il receiver type valore.

Qui chiamiamo i 2 metodi definiti per la nostra struct.

Le conversioni tra puntatori e valori per le chiamate a metodi vengono effettuate automaticamente da Go. Potresti voler usare un puntatore come receiver type per evitare di copiare nelle chiamate al metodo o per permettere al metodo di modificare il valore originale della variabile.

```go
type rect struct {
    width, height int
}
func (r *rect) area() int {
    return r.width * r.height
}
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}
func main() {
    r := rect{width: 10, height: 5}

    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())

    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```

```
area:  50
perim: 30
area:  50
perim: 30
```

INTERFACCE:

Le ***Interfacce*** in Go sono collezioni di firme (signature) di metodi.

Questa è una semplice interfaccia per le forme geometriche.

Per il nostro esempio implementeremo questa interfaccia per le struct rect e circle.

Per implementare un interfaccia in Go è sufficiente implementare tutti i metodi dell'interfaccia. Qui stiamo implementando l'interfaccia geometry per il tipo rect.

Qui invece l'implementazione per circle.

Se una variabile ha il tipo di un'interfaccia possiamo invocare i metodi dell'interfaccia stessa. Qui è presente una funzione measure generica che funzionerà su ogni variabile di tipo geometry.

Sia circle che rect implementano l'interfaccia geometry, possiamo quindi passare alla funzione measure istance di queste due struct.

```go
type geometry interface {
    area() float64
    perim() float64
}

type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}

func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}
    measure(r)
    measure(c)
}
```

```
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

ERRORI:

In Go si è soliti comunicare ***errori*** attraverso un valore restituito esplicito e separato. Questo diversamente sia da quanto accade con le eccezioni usate in linguaggi come Java e Ruby che dall'overloading di un singolo valore come risultato / errore che è a volte usato in C. L'approccio usato da Go rende facile capire quali funzioni possono generare errori e gestirli attraverso gli stessi costrutti utilizzati per qualsiasi altra attività che non riguardi la gestione degli errori.

Per convenzione, gli errori sono l'ultimo valore restituito, ed il loro tipo è error, un'interfaccia built-in.
errors.New crea un errore base con il messaggio di errore dato.

Un nil nella posizione dell'errore indica che non vi è stato alcun errore.

È possibile usare altri tipi come error implementando il metodo Error() su di essi. Di seguito una variante del precedente esempio che usa un tipo a sé per rappresentare esplicitamente un errore di parametri.

In questo caso, usiamo la sintassi &argError per costruire una nuova struct ed inserire i valori dei due campi, arg e prob.

I due cicli che seguono provano ognuna delle nostre funzioni che restituiscono un errore. Nota che l'uso di un controllo di errori sulla stessa linea dell'if è una pratica comune in Go.

Se vuoi utilizzare qualche campo specifico di un errore personalizzato, devi convertire l'errore in un'istanza dell'errore personalizzato tramite un type assertion.

```go
func f1(arg int) (int, error) {
    if arg == 42 {
        err := errors.New("impossibile calcolare con 42")
        return -1, err
    }
    return arg + 3, nil
}

type argError struct {
    arg  int
    prob string
}
func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.prob)
}

func f2(arg int) (int, error) {
    if arg == 42 {
        return -1, &argError{arg, "impossibile calcolare"}
    }
    return arg + 3, nil
}

func main() {
    for _, i := range []int{7, 42} {
        if r, e := f1(i); e != nil {
            fmt.Println("f1 ha fallito   :", e)
        } else {
            fmt.Println("f1 ha funzionato:", r)
        }
    }
    for _, i := range []int{7, 42} {
        if r, e := f2(i); e != nil {
            fmt.Println("f2 ha fallito   :", e)
        } else {
            fmt.Println("f2 ha funzionato:", r)
        }
    }
    _, e := f2(42)
    if ae, ok := e.(*argError); ok {
        fmt.Println(ae.arg)
        fmt.Println(ae.prob)
    }
}
```

```
f1 ha funzionato: 10
f1 ha fallito   : impossibile calcolare con 42
f2 ha funzionato: 10
f2 ha fallito   : 42 - impossibile calcolare
42
impossibile calcolare
```

GOROUTINE

Una ***goroutine*** è di fatto un *green* ***thread*** che permette di rendere concorrente l'esecuzione di un programma. Un *green thread* o thread virtuale sono thread schedulati da una libreria di runtime o da una macchina virtuale (VM) anziché in modo nativo dal sistema operativo (SO) sottostante.

La prima chiamata di funzione *stampa("Sincrono")* viene eseguita in modo sincrono.

La seconda e terza chiamata di funzione vengono eseguite in due goroutine separate in modo asincrono.

La funzione di Scanln richiede di immettere un input per far terminare l'esecuzione del programma, in questo modo le goroutine hanno tutto il tempo per essere eseguite e terminare la loro esecuzione.

Quando si esegue questo programma si può notare prima l'output della (prima) chiamata bloccante. Successivamente le due goroutine lanciate possono produrre dell'output "disordinato". Questo disordine riflette l'esecuzione concorrente delle goroutine da parte del runtime di Go.

```go
func stampa(parola string) {
    for i := 0; i < 2; i++ {
        fmt.Println(i, ":", parola)
    }
}

func main() {
    stampa("Sincrono")

    go stampa("Asincrono")
    go fmt.Println("Ciao")

    var input string
    fmt.Scanln(&input)
}
```

```
0 : Sincrono
1 : Sincrono
0 : Asincrono
1 : Asincrono
Ciao
```

```
0 : Sincrono
1 : Sincrono
Ciao
0 : Asincrono
1 : Asincrono
```

CHANNEL:

I ***channel*** sono delle ***pipe*** che connettono goroutine concorrenti. Una pipe è uno degli strumenti di comunicazione tra processi offerti dal sistema operativo. È possibile mandare e ricevere dei valori da una goroutine a un'altra attraverso i channel.

- Per ***creare*** un nuovo channel si usa la sintassi "*make(chan type)*". I channel sono dichiarati con i tipi dei valori che possono veicolare.
- Per ***inviare*** un valore in un channel si usa la sintassi "*nameChannel* <- ". Inviamo la stringa "ping" al channel canale che abbiamo dichiarato in precedenza, in una nuova goroutine.
- Per ***ricevere*** un valore da un channel si usa la sintassi "<- *nomeChannel*". Riceviamo il messaggio "ping" che abbiamo inviato al canale precedentemente nella goroutine e lo stamperemo.

Quando il codice viene eseguito, la stringa "ping" è passata da una goroutine all'altra attraverso il channel. Di default, l'invio e il ricevimento si bloccano finché sia il mittente che il destinatario non sono pronti. Questo ha permesso di aspettare la fine del programma per il messaggio "ping" senza dover effettuare alcuna sincronizzazione.

```go
func invio(canale chan string) {
    canale <- "ping"
}

func main() {
    canale := make(chan string)

    go invio(canale)

    msg := <-canale
    fmt.Println(msg)
}
```

CHANNEL CON BUFFER:

Per default i channel vengono creati senza ***buffer***. Scrivere un valore in un channel "chan <-" sarà quindi ***bloccante*** se non c'è nessun ricevitore in ascolto su quel channel "<- chan". Quando alcuni dati vengono scritti nel canale, la goroutine viene bloccata finché un'altra goroutine non li legge da quel canale. I ***channel con buffer*** accettano un numero limitato di valori e non risultano essere bloccanti se nessuno sta ascoltando sul channel in questione.

Creiamo un channel di stringhe con make con un buffer di dimensione 2 (se non specificata sarà 0 e quindi senza buffer). Dato che il channel è bufferizzato, è possibile inviare 1 o più valore (specificati nella funzione make) sul channel senza che il processo si blocchi.

```go
func main() {
    canale := make(chan string, 2)

    canale <- "ping1"
    canale <- "ping2"
    fmt.Println(<-canale)
    fmt.Println(<-canale)
}
```

```
ping1
ping2
```

NOTA: Un canale senza buffer necessita di un ricevitore (appartenente ad un'altra goroutine) non appena un messaggio viene inviato al canale, se non viene specificato il programma andrà in deadlock.

```go
func main() {
    canale := make(chan string)

    canale <- "ping1"
    fmt.Println(<-canale)
}
```

```
fatal error: all goroutines are asleep - deadlock!
```

CHANNEL SINCRONIZATI:

È possibile usare i ***channel*** per ***sincronizzare*** l'esecuzione attraverso delle goroutine. Usiamo un ricevimento bloccante per aspettare che una goroutine termini il suo lavoro.

La funzione lavoratore verrà eseguita da una goroutine differente. Il canale verrà usato per dire alla goroutine principale che ha finito il suo lavoro, inviando true non appena finito.

Nel main viene creato il canale da notificare una volta svolto il lavoro. Viene avviata una nuova goroutine che eseguirà la funzione definita in precedenza passandogli il canale appena creato.
L'ultima istruzione di stampa blocca l'esecuzione finché non si riceve un valore dal canale, ovvero il valore true che conferma che l'altra goroutine ha finito il lavoro.

```go
func lavoratore(canale chan bool) {
    fmt.Println("\nInizio lavoro...")
    for i := 3; i > 0; i-- {
        time.Sleep(time.Second)
        fmt.Println(i)
    }
    canale <- true
}

func main() {
    canale := make(chan bool, 1)

    go lavoratore(canale)
    fmt.Println("Lavoro finito?", <-canale)
}
```

NOTA: Se si rimuove <- *canale*, allora il programma sarebbe terminato addirittura prima che la goroutine lavoratore fosse avviata, e stamperebbe solamente "`Lavoro finito? `"

```
Inizio lavoro...
3
2
1
Lavoro finito? true
```

CANALI DIREZIONATI:

Quando si passa un channel come parametro di funzione, si può indicare se il channel deve essere utilizzato solamente per ricevere o per inviare messaggi. Questa indicazione aumenta la ***type-safety*** dell'intero programma.

La funzione emettitore accetta un channel per il solo invio "`chan<-`", se si tenta di ricevere un valore su questo canale viene generato un errore di compile-time.

La funzione ricevitore accetta un channel per la sola ricezione "`<-chan`".

```go
func emettitore(canale chan<- string, msg string) {
    canale <- msg
}
func ricevitore(canale <-chan string) {
    msg := <-canale
    fmt.Println(msg, " ...ricevuto")
}

func main() {
    canale := make(chan string, 1)

    emettitore(canale, "Messaggio sicuro")
    ricevitore(canale)
}
```

```
Messaggio sicuro  ...ricevuto
```

SELECT:

Select permette di elaborare valori ricevuti da più canali con una sola goroutine. Combinare goroutine e canali con select è uno dei punti di forza di Go.

Innanzitutto, creiamo due canali, ognuno di loro riceverà un valore con l'esecuzione delle rispettive goroutine.

Utilizzeremo select per ricevere dai due canali simultaneamente, stampando ogni valore che arriva.

NOTA: Il tempo totale di esecuzione si aggira a 2 secondi piuttosto che a 3, perché i due Sleep sono eseguiti concorrentemente.

```go
func invio1(canale1 chan string) {
    time.Sleep(1 * time.Second)
    canale1 <- "Messaggio1"
}
func invio2(canale2 chan string) {
    time.Sleep(2 * time.Second)
    canale2 <- "Messaggio2"
}

func main() {
    canale1 := make(chan string)
    canale2 := make(chan string)

    go invio1(canale1)
    go invio2(canale2)

    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-canale1:
            fmt.Println(msg1, "ricevuto")
        case msg2 := <-canale2:
            fmt.Println(msg2, "ricevuto")
        }
    }
}
```

```
Messaggio1 ricevuto
Messaggio2 ricevuto
```

OPERAZIONI SU CANALI NON BLOCCANTI:

Le operazioni di invio e di ricezione nei channel sono bloccanti. Tuttavia, possiamo usare un select con una clausola per implementare invii, ricezioni e select a più clausole non bloccanti.

Di seguito una ricezione non bloccante. Se un valore è disponibile su `messaggi`, allora select userà il case apposito. Se nessun valore è disponibile, allora selezionerà immediatamente il branch default.

Un invio non bloccante funziona più o meno alla stessa maniera.

Possiamo usare case multipli sopra la clausola default per implementare un select a più clausole non bloccanti. Qui facciamo una ricezione non-bloccante sia su messaggi che su segnali.

```go
func main() {
    messaggi := make(chan string)
    segnali := make(chan bool)

    select {
    case msg := <-messaggi:
        fmt.Println("messaggio ricevuto", msg)
    default:
        fmt.Println("nessun messaggio ricevuto")
    }

    msg := "ciao"
    select {
    case messaggi <- msg:
        fmt.Println("messaggio inviato", msg)
    default:
        fmt.Println("nessun messaggio inviato")
    }

    select {
    case msg := <-messaggi:
        fmt.Println("messaggio ricevuto", msg)
    case sig := <-segnali:
        fmt.Println("segnale ricevuto", sig)
    default:
        fmt.Println("nessuna attività")
    }
}
```

```
nessun messaggio ricevuto
nessun messaggio inviato
nessuna attività
```

RANGE SUI CHANNEL:

In un esempio precedente, abbiamo visto come ***for*** e ***range*** ci diano la possibilità di iterare su strutture di dati semplici. Possiamo usare la stessa sintassi anche per iterare sui valori ricevuti da un channel.

Itereremo su 2 valori nel canale queue.

Questo range reitera su ogni elemento ogni qual volta che viene ricevuto dal channel queue. Poiché abbiamo effettuato un close sul canale in precedenza, la iterazione terminerà dopo aver ricevuto i 2 elementi.

NOTA: Se non avessimo effettuato il close sul canale, allora saremmo bloccati ad attendere per un terzo futuro valore.

```go
func main() {
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)

    for elem := range queue {
        fmt.Println(elem)
    }
}
```

```
one
two
```

TIMEOUT:

I ***Timeout*** sono fondamentali per i programmi che si connettono a risorse esterne o che hanno comunque bisogno di limitare il tempo d'esecuzione. Implementare i timeout in Go è semplice grazie ai channel e al costrutto select.

Ai fini del nostro esempio, supponiamo di star eseguendo una chiamata esterna che restituisce il suo risultato sul channel c1 dopo 2 secondi.

Qui mostriamo il select che implementa un timeout. Il comando `res := <-canale1` attende il risultato della funzione mentre il comando `<-time.After()` emette un risultato dopo un timeout di 1 secondo. Dal momento che il comando select procede con il primo canale per cui è disponibile un valore, eseguiremo il caso del timout se la nostra chiamata esterna richiede più di 1 secondo.

In questo caso possiamo vedere come impostare un timeout di 3 secondi sia sufficiente a far restituire la nostra goroutine che scrive sul canale `canale2`. Riusciremo infatti a vedere il risultato a schermo e a non far scattare il timeout.

Se si esegue questo programma si potrà notare come la prima operazione andrà in timeout, mentre la seconda verrà eseguita correttamente.

```go
func main() {
    canale1 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        canale1 <- "risultato 1"
    }()

    select {
    case res := <-canale1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout 1")
    }

    canale2 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        canale2 <- "risultato 2"
    }()
    select {
    case res := <-canale2:
        fmt.Println(res)
    case <-time.After(time.Second * 3):
        fmt.Println("timeout 2")
    }
}
```

```
timeout 1
risultato 2
```

CHIUDERE UN CANALE:

Chiudere un channel serve ad indicare che non verranno più inviati valori su quel channel. Questo è utile per comunicare il completamento di un operazione a chi sta ascoltando sul channel.

In questo esempio useremo un channel jobs per inviare i task che devono essere eseguiti dalla goroutine main() a una goroutine che svolge il ruolo di worker. Quando non si hanno più task da eseguire si chiuderà il channel jobs tramite il built-in close.

Questa è la goroutine worker, che riceverà i task in modo continuativo dal channel jobs tramite il comando `j, more := <-jobs`. Utilizzando questa forma speciale per ricevere i valori, nella variabile `more` avremo il valore *false* se il channel jobs è stato chiuso e tutti i valori sono stati ricevuti. Utilizzeremo questo valore per indicare quando la goroutine ha terminato di eseguire tutti i task che ha ricevuto.

Questo invia 3 job alla goroutine worker sul channel jobs e lo chiude subito dopo.

Attendiamo la goroutine worker utilizzando l'approccio di sincronizzazione.

```go
func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)

    go func() {
        for {
            j, more := <-jobs
            if more {
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true
                return
            }
        }
    }()
    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }
    close(jobs)
    fmt.Println("sent all jobs")

    <-done
}
```

```
sent job 1
sent job 2
sent job 3
sent all jobs
received job 1
received job 2
received job 3
received all jobs
```

WORKER POOL:

In questo esempio vedremo come implementare un ***worker pool*** usando le goroutine e i channel.

Questo è il worker, sul quale eseguiremo i nostri task concorrenti. Questi worker riceveranno i task da eseguire sul channel jobs ed invieranno i risultati sul channel results. In questo esempio abbiamo inserito una Sleep da un secondo per simulare un task oneroso.

Per poter utilizzare il pool di worker dobbiamo poter inviare i task e poter ricevere i risultati, creiamo dunque due channel per questo.

Questo farà partire 2 worker, che saranno inizialmente bloccati in quanto non hanno task da eseguire.

Inviamo 6 task da eseguire sul channel job ed invochiamo la close su quel canale, in modo da indicare che non ci sono altri task da eseguire.

Infine, recuperiamo i risultati delle computazioni sul channel results.

```go
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "sta eseguendo task", j)
        time.Sleep(time.Second)
        results <- j
    }
}
func main() {
    jobs := make(chan int, 100)
    results := make(chan int, 100)

    for w := 1; w <= 2; w++ {
        go worker(w, jobs, results)
    }

    for j := 1; j <= 6; j++ {
        jobs <- j
    }
    close(jobs)

    for a := 1; a <= 6; a++ {
        fmt.Println("worker", <-results, "terminato")
    }
}
```

Il nostro programma in esecuzione mostra i 6 task che vengono eseguiti dai vari worker. Il programma termina in circa 2 secondi, anche se il totale dei task avrebbe richiesto 6 secondi. Questo perché ci sono 2 worker che vengono eseguiti in parallelo.

```
worker 2 sta eseguendo task 1
worker 1 sta eseguendo task 2
worker 1 sta eseguendo task 3
worker 2 terminato
worker 2 sta eseguendo task 4
worker 1 terminato
worker 1 sta eseguendo task 5
worker 2 sta eseguendo task 6
worker 3 terminato
worker 4 terminato
worker 6 terminato
worker 5 terminato
```

WAITGROUPS:

Per attendere la fine di più goroutine, possiamo utilizzare un ***gruppo di attesa***.

La funzione worker verrà eseguita in ogni goroutine. La funzione Sleep simulerà un processo.

La WaitGroup è usata per aspettare che tutte le goroutine avviate finiscano la loro esecuzione. **NOTA**: se una WaitGroup venisse passata esplicitamente alle funzioni, dovrebbe essere fatto utilizzando un puntatore.

Il for lancia diverse goroutine e incrementa il contatore della WaitGroup per ognuna. **NOTA**: si evita il riuso del valore "i" in ogni chiusura goroutine.

Si racchiude la chiamata al worker in una chiusura per assicurare di comunicare a WaitGroup che questo worker ha terminato, tramite funzione Done() che decrementa il contatore. In questo modo il worker stesso non deve essere consapevole delle primitive di concorrenza coinvolte nella sua esecuzione.

Viene utilizzata la Wait() sul gruppo per aspettare la terminazione delle varie goroutine. Questa funzione aspetta che il contatore arrivi a 0.

NOTA: questo approccio non ha un modo semplice per propagare gli errori dai worker. Per gestire questi errori esiste il pacchetto *errgroup*.

```go
func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}
func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 3; i++ {
        wg.Add(1)
        i := i
        go func() {
            defer wg.Done()
            worker(i)
        }()
    }
    wg.Wait()
}
```

```
Worker 2 starting
Worker 3 starting
Worker 1 starting
Worker 1 done
Worker 2 done
Worker 3 done
```

DEFER:

Il built-in ***defer*** può essere utilizzato per assicurarci che una funzione venga eseguita in un secondo momento, ad esempio per scopi di pulizia o per rilasciare delle risorse. Può essere assimilata alle keywork ensure o ***finally*** di altri linguaggi.

Supponiamo che venga avviato un programma, esso richiede delle risorse, successivamente queste devono essere rilasciate.

Utilizzando defer su una operazione, essa viene eseguita al termine della funzione dentro la quale è stata chiamata (in questo caso main).

La stampa di "Rilascio risorse…" verrà eseguita subito dopo l'ultima operazione del main, in tal caso dopo l'ultima stampa.

```go
func main() {
    fmt.Println("Inizio programma...")

    defer fmt.Println("Rilascio risorse...")

    time.Sleep(time.Second * 3)
    fmt.Println("Esecuzione terminata...")
}
```

```
Inizio programma...
Esecuzione terminata...
Rilascio risorse...
```

JSON:

Go ha il supporto built-in per l'encoding e il decoding di ***JSON***, con anche la possibilità di convertire verso o da tipi built-in e personalizzati.

Useremo questi due struct per dimostrare l'encoding e il decoding di tipi personalizzati.

Come prima cosa vedremo come fare l'encoding di datatype nativi per ottenere stringhe JSON. Ecco degli esempi per dei valori atomici.

E ora degli esempi con le slice e le map, che restituiscono degli array e oggetti JSON come ti aspetteresti.

Il package JSON può codificare automaticamente dei data type personalizzati, ad esempio delle struct. Includerà solo i field esportati nell'output codificato e di default userà i nomi dei field per le chiavi JSON.

Puoi usare dei tag sulle dichiarazioni dei field di una struct per personalizzare il nome della chiave nel JSON finale. Vedi la dichiarazione di Response2 sopra per vedere un esempio di questi tag.

Ora diamo un'occhiata al decoding di valori JSON in valori builtin di Go. Di seguito un esempio di una struttura di dati generica.

Abbiamo bisogno di definire una variabile dove il package JSON metterà i dati decodificati. Questa map[string]interface{} sarà una map di stringhe che puntano a dati con datatype arbitrario.

Qui avviene il vero decoding, dove controlleremo anche se c'è stato un errore.

Per poter usare i dati nella mappa decodificata, avremo bisogno di fare un type assertion al loro vero valore. Ad esempio, qui facciamo un type assertion del valore in num in un float64.

Accedere a dati annidati richiede una serie di type assertion.

Possiamo anche fare il decode di JSON in data type personalizzati. Questo ha il vantaggio di aggiungere un'ulteriore type-safety ai nostri programmi e elimina la necessità dell'uso dei type assertion quando accediamo a dati decodificati.

Negli esempi precedenti abbiamo sempre usato i byte e le stringhe come tramiti tra i dati e la rappresentazione JSON sullo standard output. Possiamo anche indirizzare la codifica JSON direttamente a degli io.Writer come os.Stdout o addirittura nei corpi delle risposte HTTP.

```go
type Response1 struct {
    Pagina int
    Frutti []string
}
type Response2 struct {
    Pagina int      `json:"pagina"`
    Frutti []string `json:"frutti"`
}

func main() {
    bolB, _ := json.Marshal(true)
    fmt.Println(string(bolB))

    intB, _ := json.Marshal(1)
    fmt.Println(string(intB))

    fltB, _ := json.Marshal(2.34)
    fmt.Println(string(fltB))

    strB, _ := json.Marshal("gopher")
    fmt.Println(string(strB))

    slcD := []string{"mela", "pesca", "pera"}
    slcB, _ := json.Marshal(slcD)
    fmt.Println(string(slcB))

    mapD := map[string]int{"mela": 5, "lattuga": 7}
    mapB, _ := json.Marshal(mapD)
    fmt.Println(string(mapB))

    res1D := &Response1{
        Pagina: 1,
        Frutti: []string{"mela", "pesca", "pera"}}
    res1B, _ := json.Marshal(res1D)
    fmt.Println(string(res1B))

    res2D := &Response2{
        Pagina: 1,
        Frutti: []string{"mela", "pesca", "pera"}}
    res2B, _ := json.Marshal(res2D)
    fmt.Println(string(res2B))

    byt := []byte(`{"num":6.13,"strs":["a","b"]}`)

    var dat map[string]interface{}

    if err := json.Unmarshal(byt, &dat); err != nil {
        panic(err)
    }
    fmt.Println(dat)

    num := dat["num"].(float64)
    fmt.Println(num)

    strs := dat["strs"].([]interface{})
    str1 := strs[0].(string)
    fmt.Println(str1)

    str := `{"pagina": 1, "frutti": ["mela", "pesca"]}`
    res := Response2{}
    json.Unmarshal([]byte(str), &res)
    fmt.Println(res)
    fmt.Println(res.Frutti[0])

    enc := json.NewEncoder(os.Stdout)
    d := map[string]int{"mela": 5, "lettuce": 7}
    enc.Encode(d)
}
```

```
true
1
2.34
"gopher"
["mela","pesca","pera"]
{"lattuga":7,"mela":5}
{"Pagina":1,"Frutti":["mela","pesca","pera"]}
{"pagina":1,"frutti":["mela","pesca","pera"]}
map[num:6.13 strs:[a b]]
6.13
a
{1 [mela pesca]}
mela
{"lettuce":7,"mela":5}
```

NUMERI CASUALI:

Il pacchetto math/rand di Go offre funzionalità per la generazione di ***numeri pseudo-casuali***.

Ad esempio, rand.Intn(100) restituisce un int n tale che 0 <= n < 100.

rand.Float64 restituisce un valore di tipo float64 f tale che 0.0 <= f < 1.0.

Questa espressione può essere utilizzata per genere float random in un altro intervallo, In questo caso5.0 <= f' < 10.0.

Il generatore di numeri casuali è deterministico, produrrà quindi la stessa sequenza di numeri ad ogni esecuzione. Per generare sequenze casuali ad ogni esecuzione, dobbiamo settare un seed variabile. Tieni in considerazione che non è sicuro utilizzare questo metodo per generare numeri che devono essere segreti. Se hai tali necessità è consigliabile utilizzare il pacchetto *crypto/rand*.

È quindi possibile invocare rand.Rand come la funzione rand del pacchetto.

Se imposti come seed lo stesso numero, vedrai che verrà generata la stessa sequenza di numeri.

```go
func main() {
    fmt.Print(rand.Intn(100), ",")
    fmt.Print(rand.Intn(100))
    fmt.Println()

    fmt.Println(rand.Float64())

    fmt.Print((rand.Float64()*5)+5, ",")
    fmt.Print((rand.Float64() * 5) + 5)
    fmt.Println()

    s1 := rand.NewSource(time.Now().UnixNano())
    r1 := rand.New(s1)

    fmt.Print(r1.Intn(100), ",")
    fmt.Print(r1.Intn(100))
    fmt.Println()

    s2 := rand.NewSource(42)
    r2 := rand.New(s2)
    fmt.Print(r2.Intn(100), ",")
    fmt.Print(r2.Intn(100))
    fmt.Println()
    s3 := rand.NewSource(42)
    r3 := rand.New(s3)
    fmt.Print(r3.Intn(100), ",")
    fmt.Print(r3.Intn(100))
}
```

```
81,87
0.6645600532184904
7.1885709359349015,7.123187485356329
18,64
5,87
5,87
```

HASH SHA1:

La ***funzione SHA-1*** viene frequentemente utilizzata per calcolare il digest (l'impronta) dei file binari o di testo. Ad esempio, il sistema di versionamento git utilizza gli SHA-1 in modo intensivo per identificare i file e le cartelle versionate. Ecco come calcolare gli hash SHA-1 in Go.

Si possono trovare implementazioni di svariati algoritmi crittografici dentro il pacchetto *crypto/**.

Il pattern per creare un nuovo hash è sha1.New(), sha1.Write(bytes), ed infine sha1.Sum([]byte{}).
Iniziamo creando un nuovo hash.
La funzione Write lavora con i bytes. Se si deve calcolare l'hash di una stringa s è possibile utilizzare []byte(s) per ottenerne i bytes.
Con la funzione Sum è possibile ottenere il valore dell'hash. L'argomento può essere utilizzato per accodare il valore calcolato ad uno slice di byte esistente, non è generalmente necessario.
Gli SHA-1 sono generalmente stampati in formato esadecimale (ad esempio nei commit di git).
Utilizza '%x' per stampare il valore in esadecimale.

Eseguendo il programma si nota che viene calcolato l'hash della funzione e stampato in esadecimale a schermo

```go
func main() {
    s := "sha1 this string"

    h := sha1.New()

    h.Write([]byte(s))

    bs := h.Sum(nil)

    fmt.Println(s)
    fmt.Printf("%x\n", bs)
}
```

```
sha1 this string
cf23df2207d99a74fbe169e3eba035e633b65d94
```

È possibile calcolare altre funzioni hash in un modo molto simile. Ad esempio, per calcolare l'MD5 è sufficiente importare crypto/md5 ed usare md5.New().

CHAINCODE IN GO (HYPERLEDGER FABRIC):

In ambito ***blockchain***, un ***Chaincode*** è un programma che implementa un'interfaccia prescritta, che viene eseguito in un processo separato dal peer e, inizializza e gestisce lo stato del libro mastro (ledger) tramite le transazioni inviate dalle applicazioni.

Un chaincode in genere gestisce la logica di business concordata dai membri della rete, simile a uno "smart contract". È possibile richiamare un chaincode per aggiornare o interrogare il libro mastro in una transazione proposta. Data l'autorizzazione appropriata, un chaincode può invocare un altro chaincode, nello stesso canale o in uno diversi, per accedere al suo stato.

NOTA: se il chaincode chiamato si trova su un canale diverso dal chaincode chiamante, è consentita solo la query di lettura. Cioè, il chaincode chiamato su un canale diverso è solo una Query, che non partecipa ai controlli di convalida dello stato nella successiva fase di commit.

CAHINCODE API:

Ogni programma chaincode deve implementare l'interfaccia ***Chaincode*** i cui metodi vengono chiamati in risposta alle transazioni ricevute. *Docs in Go*.

Il metodo ***Invoke*** viene chiamato dai clienti per presentare proposte di transazione. Questo metodo consente di utilizzare il chaincode per leggere e scrivere dati sul libro mastro del canale. Bisogna prima, però includere un metodo ***Init*** che fungerà da funzione di inizializzazione per il chaincode. Questo metodo verrà chiamato per inizializzare il chaincode quando viene avviato o aggiornato. Per impostazione predefinita, questa funzione non viene mai eseguita. Tuttavia, è possibile utilizzare la definizione del chaincode per richiedere l'esecuzione della funzione *Init*. Se viene richiesta l'esecuzione di *Init*, fabric farà in modo che venga invocato prima di qualsiasi altra funzione e venga invocato solo una volta. Questa opzione fornisce un controllo aggiuntivo su quali utenti possono inizializzare il chaincode e la possibilità di aggiungere dati iniziali al libro mastro.

L'altra interfaccia nelle API "shim" del chaincode è ***ChaincodeStubInterface***, che viene utilizzato per accedere e modificare il libro mastro e per effettuare chiamate tra i chaincode.

ESEMPIO CHAINCODE IN GO:

Prima di tutto aggiungiamo una struttura *SimpleAsset* come ricevitore per le funzioni di shim Chaincode.

```
type SimpleAsset struct {
}
```

Successivamente, implementeremo la funzione Init. Recupereremo gli argomenti della chiamata Init utilizzando la funzione *GetStringArgs()* e verificheremo la validità. Nel nostro caso, ci aspettiamo una coppia chiave-valore.

Ora che abbiamo stabilito che la chiamata è valida, memorizzeremo lo stato iniziale nel libro mastro. Per fare ciò, chiameremo *PutState()* con la chiave e il valore passati come argomenti. Supponendo che tutto sia andato bene, restituisci un oggetto *peer.Response* che indica che l'inizializzazione è riuscita.

```
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }

  err := stub.PutState(args[0], []byte(args[1]))
  if err != nil {
    return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
  }
  return shim.Success(nil)
}
```

Come con la funzione Init, dobbiamo estrarre gli argomenti dal file ChaincodeStubInterface. Gli argomenti della funzione Invoke saranno il nome della funzione dell'applicazione chaincode da invocare. Nel nostro caso, l'applicazione avrà semplicemente due funzioni: set e get, che consentono di impostare il valore di un asset o di recuperare il suo stato attuale. Per prima cosa chiamiamo *GetFunctionAndParameters()* per estrarre il nome della funzione e i parametri per quella funzione dell'applicazione chaincode.

Successivamente, convalideremo il nome della funzione come set o get, e invocheremo quelle funzioni dell'applicazione chaincode, restituendo una risposta appropriata tramite le funzioni shim.Success o shim.Error che serializzeranno la risposta in un messaggio protobuf gRPC.

```
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else {
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    return shim.Success([]byte(result))
}
```

L'applicazione chaincode implementa due funzioni che possono essere invocate tramite la funzione Invoke. Implementiamo queste funzioni ora:

- *Set*, memorizza l'asset (chiave e valore) nel libro mastro. Se la chiave esiste, sovrascriverà il valore con quello nuovo;
- *Get*, restituisce il valore della chiave asset specificata.

```
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
            return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
    }
    if value == nil {
            return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}
```

Infine, dobbiamo aggiungere la funzione main, che chiamerà la funzione shim.Start.

```
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
            fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}
```