# Go

1. **Go is statically typed**: Statically typed is a programming language characteristic in which variable types are explicitly declared and thus are determined at compile time
2. **GO is** compiled type high-level programming language

## Variable Declaration

1. Using var keyword:
2. var variable\_name type = value

var name string = “shuvo”

1. var num = 5
2. Short variable declaration (without var) :

Shuvo := name

**Code:**

package main

import "fmt"

func main() {

var num1 = 5 // type inferred

var num2 int // explicitly typed

fmt.Println(num1)

fmt.Println(num2)

}

1. const :

const name = “shuvo”

## Data Type

1. Basic type
2. Composite type
3. Basic type:
   1. Numeric type:
      1. Integers
      2. Floats
      3. Complex Numbers
   2. String:
      1. String always in “” quotation.
   3. Byte:
   4. Rune: it use Unicode character.
   5. Boolean
4. Composite type:
   1. Aggregate type/Non-Reference type:
      1. Array
      2. Structure
   2. Reference type
      1. Pointer
      2. Slices
      3. Maps
      4. Function
      5. Channels
   3. Interface type

## Type Conversions

1. Use the %T printing verb in the Printf() function.

fmt.Printf("%T\n", start) // time.Time

1. Use the reflect package. The reflect package allows you to find out the data type of a variable.

fmt.Println(reflect.TypeOf(start)) // time.Time

fmt.Println(reflect.ValueOf(start).Kind()) // struct

## Flow Control

1. **If/Else:**

num := 5

condition := num % 2 == 1

if condition {

fmt.Println("Number is odd")

}

**Short-circuiting:**

Go evaluates conditions using a method known as short-circuiting.

func raining() bool {

fmt.Println("Check if it is raining now...")

return true

}

func snowing() bool {

fmt.Println("Check if it is snowing now...")

return true

}

if raining() || snowing() {

fmt.Println("Stay indoors!")

}

1. **Switch:**

**Switching with fall-throughs:**

grade := "C"

switch grade {

case "A":

fallthrough

case "B":

fallthrough

case "C":

fallthrough

case "D":

48 PART 1 Getting Started with Go

fmt.Println("Passed")

case "F":

fmt.Println("Failed")

default:

fmt.Println("Absent")

}

**Matching multiple cases:**

grade := "C"

switch grade {

case "A", "B", "C", "D":

fmt.Println("Passed")

case "F":

fmt.Println("Failed")

default:

fmt.Println("Undefined")

}

1. **For:**
2. **For- range:**

## Function

1. **Access modifier:**
2. First letter of function Capital indicate that function is public.
3. First letter small indicate private function
4. **Multiple value return:**

Func square(num float64) float64 int{

Return x,y

}

1. **Variadic function:**

func myFunc(numbers ...int) {

fmt.Println(numbers)

}

1. **Defer:**
2. **Named Result Parameter**
3. **Anonymous Functions:**

var i func()

int i = func() int {

return 5

}

func main() {

var i func()

int i = func() int {

return 5

}

fmt.Println(i()) // 5

}

1. **Closure anonymous functions:**

Closure: a function value that references variables from outside its body

func fib() = func() int {

f1 := 0

f2 := 1

return func() int {

f1, f2 = f2, (f1 + f2)

return f1

}

}

func main() {

gen := fib()

fmt.Println(gen()) // 1

}

1. **Invoke anonymous function:**

“()” This part immediately invokes (calls) the function. The empty parentheses indicate that the function is called without any arguments.

go func(){

}()

1. **Implementing the filter() function using closure**
2. The filter() function takes in a collection of items and returns another collection containing the items you want.
3. The map() function allows you to “map” items from one collection into another collection.
4. The reduce() function returns a single value based on the collection you pass in

**CODE:**

func filter(arr []int, cond func(int) bool) []int {

result := []int{}

for \_, v := range arr {

if cond(v) {

result = append(result, v)

}

}

return result

}

func main() {

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

evens := filter( a , func(val int) bool { return val%2 == 0 } )

fmt.Println(evens)

}

## Array

## Slice

## Map

1. A map is a hash table that stores data in an associative manner. Items in a map are not accessed according to their positions. Instead, you use keys (a set of unique value that identifies the elements in a map).
2. The problem with our program is that maps have to be initialized before they can be used.
3. maps are not sequential.

syntax:

map[keyType] valueType

code:

package main

var heights map[string]int

func main() {

}

**Create map**

1. with make() :

Because the map type is a reference type, you need to first initialize it using the make() function before you can use it:

var heights map[string]int

func main() {

heights = make(map[string]int)

}

1. with a map literal:

heights := map[string]int{

"Peter": 170,

"Joan": 168,

"Jan": 175, // <-- note the comma here

}

**Checking the existence of a key:**

if v, ok := heights["Jim"]; ok {

fmt.Println(v)

} else {

fmt.Println("Key does not exist")

}

the heights variable returns two values: the value of the specified key, as well as a Boolean value (stored in ok) indicating if the operation succeeded. If ok is true, the specified key exists, and you can go ahead and use its value (stored in v). If ok is false, the key does not exist

**Deleting a key:**

delete(map, key)

if \_, ok := heights["Joan"]; ok {

delete(heights, "Joan")

} else {

fmt.Println("Key does not exist")

}

**number of items in a map:**

fmt.Println(len(heights))

**Iterating over a map:**

for k, v := range heights {

fmt.Println(k, v)

}

**Sort map:**

sort.Strings(keys)

fmt.Println(keys)

## Struct

**Syntax**:

type StructName struct {

Field1 datatype

Field2 datatype

}

**Code**:

type Player struct {

name string

age int

}

**Unexported**: The first word of variable should be small. We can access this variable from the other struct.

type Player struct {

city string // Unexported

state string

}

**Exported**: the first word of variable should be Capital. We cannot access this variable from the other struct.

type Player struct {

Name string // Exported

Age int

}

**Declared process:**

type Circle struct {

    x float64

    y float64

    r float64

}

func main() {

    var c Circle

    c.x = 1

    c.y = 2

    c.r = 3

fmt.Println(c) // {1 2 3}

    m := new(Circle)

    m.x = 7

    m.r = 10

    m.y = 11

fmt.Println(m) // &{7 11 10}

    p := Circle{1, 5, 9}

    fmt.Println(p) // {1 5 9}

}

## Method

**1. How method come:**

* We make a struct then we can pass this struct to a function.

type Circle struct {

    x float64

    y float64

}

func Area(c Circle) float64 {

    return c.x \* c.y

}

func main() {

    m := Circle{1, 2}

    Area(m)

}

* We can also send address of struct in function.

type Circle struct {

    x float64

    y float64

}

func Area(c \*Circle) float64 {

    return c.x \* c.y

}

func main() {

    m := Circle{1, 2}

    Area( &m)

}

This way can make function to method.

**Method:**

type Circle struct {

    x float64

    y float64

}

func (c \*Circle) Area() float64 {

    c.x = 10

    c.y = 5

    return c.x \* c.y

}

func main() {

var c Circle

c.x = 1

c.y = 2

c.Area();

}

Struct represent the HAS-A relationship.

**Embedded Type relation:**

Go support IS-A relation by embedded type:

type Person struct {

Name string

}

func (p \*Person) Talk() {

fmt.Println("Hi, my name is", p.Name)

}

type Android struct {

Person

Model string

}

func main(){

a := new(Android)

a.Person.Talk()

}

## Interface

An interface defines the behavior of an object, specifying the methods that it needs to implement. Interfaces serve two important purposes in Go:

1. They make your code more versatile.
2. They force you to adopt code encapsulation (the practice of hiding the implementation of your methods).

Syntax:

type name interface {

method declear……

}

Implementing an interface:

1. **Stringer interface:**

The Stringer interface is a type that describes itself as a string. , how does the fmt.Println() function know how to format the output of the object or variable that it’s trying to print? Turns out that the fmt package defines an interface called Stringer.

When you try to print some objects using the fmt.Println() function, the function looks at this interface to see how to print the value of the specified object. So, if you don’t like the way the Person struct is printed out:

type Stringer interface {

String() string

}

Overriding the default behavior for the struct’s string() function. Implement your own Stringer interface’s string() function, like this

type Person struct {

FirstName string

LastName string

Age int

}

func (p Person) String() string {

return fmt.Sprintf("%v %v ,p.FirstName, p.LastName,)

}

1. **Empty interface:**

## Json

### Decoding JSON:

The json.Unmarshal() function parses the JSON-encoded data. If the unmarshalling (decoding) works, the json.Unmarshal() function returns a nil,

Code:

type People struct {

Firstname string

Lastname string

}

func main() {

var person People

jsonString := `{"firstname":"Wei-Meng", "lastname":"Lee"}`

json.Unmarshal([]byte(jsonString), &person)

}

**Mapping custom attribute names**

type Rates struct {

Base string `json:"base currency"`

Symbol string `json:"destination currency"`

}

func main() {

jsonString := `{

"base currency":"EUR",

"destination currency":"USD"

}`

var rates Rates

json.Unmarshal([]byte(jsonString), &rates)

fmt.Println(rates.Base) // EUR

fmt.Println(rates.Symbol) // USD

}

**Mapping unstructured data:**

func main() {

jsonString := `{

"success": true,

"timestamp": 1588779306,

"base": "EUR",

"date": "2020-05-06",

"rates": {

"AUD": 1.683349,

"CAD": 1.528643,

"GBP": 0.874757,

"SGD": 1.534513,

"USD": 1.080054

}

}`

var result map[string]interface{}

json.Unmarshal([]byte(jsonString), &result)

fmt.Println(result["success"])

rates := result["rates"]

fmt.Println(rates)

currencies := rates.(map[string]interface{})

SGD := currencies["SGD"]

fmt.Println(SGD)

}

### Encoding struct to JSON

The Marshal() function returns the encoded JSON (in a slice of bytes) and error (if any). To indent the output and format it nicely, you can use the json. MarshalIndent() function:

type Name struct {

FirstName string

LastName  string

}

type Address struct {

Line1 string

Line2 string

}

type Customer struct {

Name    Name

Address Address

}

func main() {

   john := Customer{

      Name: Name{

FirstName: "John",

          LastName: "Smith",

      },

      Address: Address{

          Line1: "The White House",

          Line2: "1600 Pennsylvania Avenue NW",

          Line3: "Washington, DC 20500",

      },

   }

   johnJSON, err := json.Marshal(john)

   if err == nil {

      fmt.Println(string(johnJSON))

   } else {

      fmt.Println(err)

   }

}

## Concurrency

Concurrency is process of multiple process as the same time.

### Threading Using Goroutine

1. Goroutine is a lightweight thread managed by the Go runtime.
2. A goroutine is a function that is capable of running concurrently with other functions.
3. To run a function as a Goroutine, simply call it using the go keyword.

func main() {

go say("Hello", 3)

go say("World", 2)

}

1. Main() function is the main goroutine of Golang. When program is terminate all goroutine is terminate immediately.

### Channels

Channels provide a way for two goroutines to communicate with one another and synchronize their execution.

Close ():

1. Only the sender should close the channel.  Closing a channel indicates that no more values will be sent on the channel, and any attempts to send on the channel will result in a panic.
2. The range loop can be used to receive values from the channel until the channel is closed. The loop will automatically terminate when the channel is closed, and any values sent before the channel was closed will be received.
3. It is possible to check if the channel is closed before receiving a value by using a comma ok idiom. The idiom returns two values, the value received from the channel and a boolean value that is true if the channel is open or false if the channel is closed.
4. If you are receiving values from multiple channels, you can use a select statement to receive values until all channels are closed. The select statement will block until a value is received from one of the channels or all channels are closed.

## Package

1. String :
2. Input/output
3. File/Folder
4. Error
5. Container
6. Sort
7. Hashes & Cryptography
8. Server
9. Http
10. RPC
11. Synchronization Primitives

### String

1. Contains:
2. Count:
3. HasPrefix
4. HasSuffix
5. Index
6. Join
7. Repeat
8. Replace
9. Split
10. ToLower
11. ToUpper

### Input / Output

1. Copy:

type Reader interface{

Read()

}

type buffer strcut{

}

### File/ Folder

1. Open();
2. Close()
3. ReadFile()
4. Create()
5. Readdir()

### Container

1. **List:** container/list package implements a doubly-linked list

func main() {

var x list.List

x.PushBack(1)

x.PushBack(2)

for e := x.Front(); e != nil; e=e.Next() {

fmt.Println(e.Value.(int))

}

}

## Net

1. http
2. internal
3. mail
4. rpc
5. smtp
6. url
7. netip

## Http

1. Clients
2. Servers
3. http2
4. request.go

type Request struct{

method string

URL \*url.URL

Proto string

ProtoMajor int

ProtoMinor int

Header Header

Body io.ReadCloser

Form url.Values

PostForm url.Values

RequestURI string

Response \*Response

GetBody func()  
}

1. func NewRequest:

func NewRequest(method, url string, body io.Reader) (\*Request, error)

1. func ReadRequest:

func ReadRequest(b \*bufio.Reader) (\*Request, error)

1. func (\*Request) FormValue

func (r \*Request) FormValue(key string) string

1. func Post:

func Post(url, contentType string, body io.Reader) (resp \*Response, err error)

## RPC

## Synchronization Primitives

# Beego

**Configuration (conf):**

Beego configuration file uses the INI format. Other supported formats include XML, JSON, and YAML.