Если вы видите что-то необычное, просто сообщите мне.

?????????? go

Golang

Summary

Hello World

```
package main

import "fmt"

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

Return to Summary

Go CLI Commands

```
# Compile & Run code
$ go run [file.go]

# Compile
$ go build [file.go]
# Running compiled file
$ ./hello

# Test packages
$ go test [folder]

# Install packages/modules
$ go install [package]

# List installed packages/modules
$ go list

# Update packages/modules
$ go fix

# Format package sources
$ go fmt
```

```
# See package documentation
$ go doc [package]

# Add dependencies and install
$ go get [module]

# See Go environment variables
$ go env

# See version
$ go version
```

Return to Summary

Go Modules

- Go projects are called **modules**
- Each module has multiple **packages**
- Each package should has a scoped functionality. Packages talk to each other to compose the code
- A module needs at least one package, the **main**
- The package main needs a entry function called **main**

```
# Create Module
$ go mod init [name]
```

Tip: By convention, modules names has the follow structure:

domain.com/user/module/package

Example: github.com/spf13/cobra

Return to Summary

Basic Types

Type	Set of Values	Values
bool	boolean	true/false
string	array of characters	needs to be inside ""
int	integers	32 or 64 bit integer
int8	8-bit integers	[-128, 128]
int16	16-bit integers	[-32768, 32767]
int32	32-bit integers	[-2147483648, 2147483647]
int64	64-bit integers	[-9223372036854775808, 9223372036854775807]
uint8	8-bit unsigned integers	[0, 255]
uint16	16-bit unsigned integers	[0, 65535]
uint32	32-bit unsigned integers	[0, 4294967295]
uint64	64-bit unsigned integers	[0, 18446744073709551615]
float32	32-bit float	
float64	64-bit float	
complex64	32-bit float with real and imaginary parts	
complex128	64-bit float with real and imaginary parts	
byte	sets of bits	alias for uint8
rune	Unicode characters	alias for int32

Return to Summary

Variables

```
// Declaration
var value int

// Initialization
value = 10
```

```
// Declaration + Initialization + Type inference
var isActive = true

// Short declaration (only inside functions)
text := "Hello"

// Multi declaration
var i, j, k = 1, 2, 3

// Variable not initialized = Zero values
// Numeric: 0
// Boolean: false
// String: ""
// Special value: nil (same as null)

var number int // 0
var text string // ""
var boolean bool // false

// Type conversions
// T(v) converts v to type T

i := 1.234 // float
int(i) // 1

// Constants
const pi = 3.1415
```

Operators

Return to Summary

Arithmetic Operators

Symbol	Operation	Valid Types
+	Sum	integers, floats, complex values, strings

Symbol	Operation	Valid Types
-	Difference	integers, floats, complex values
*	Product	integers, floats, complex values
/	Quotient	integers, floats, complex values
%	Remainder	integers
&	Bitwise AND	integers
`	`	Bitwise OR
^	Bitwise XOR	integers
&^	Bit clear (AND NOT)	integers
<<	Left shift	integer << unsigned integer
>>	Right shift	integer >> unsigned integer

Comparison Operators

Symbol	Operation
==	Equal
!=	Not equal
<	Less
<=	Less or equal
>	Greater
>=	Greater or equal

Logical Operators

Symbol	Operation
&&	Conditional AND
`	
!	NOT

Return to Summary

Conditional Statements

```
// If / Else
i := 1

if i > 0 {
    // Condition is True! i is greater than zero
} else {
    // Condition is False! i is lower or equal to zero
}

// Else if
i := 1

if i > 0 {
    // Condition is True! i is greater than zero
} else if i > 0 && i < 2 {
    // Condition is True! i greater than zero and lower than two
} else if i > 1 && i < 4 {
    // Condition is True! i greater than one and lower than four
} else {
    // None of the above conditions is True, so it falls here
}

// If with short statements
i := 2.567

if j := int(i); j == 2 {
    // Condition is True! j, the integer value of i, is equal to two
} else {
    // Condition is False! j, the integer value of i, is not equal to two
}

// Switch
text := 'hey'

switch text {
    case 'hey':
        // 'Hello!'
```

```
    case 'bye':
        // 'Byee'
    default:
        // 'Ok'
}

// Switch without condition
value := 5

switch {
    case value < 2:
        // 'Hello!'
    case value >= 2 && value < 6:
        // 'Byee'
    default:
        // 'Ok'
}
```

Return to Summary

Loops

```
// Golang only has the for loop
for i := 0; i < 10; i++ {
    // i
}

// The first and third parameters are ommitable
// For as a while
i := 0;

for i < 10 {
    i++
}

// Forever loop
for {
```

```
}
```

Return to Summary

Arrays

```
// Declaration with specified size
var array [3]string
array[0] = "Hello"
array[1] = "Golang"
array[2] = "World"

// Declaration and Initialization
values := [5]int{1, 2, 3, 4, 5}

// Slices: A subarray that acts as a reference of an array
// Determining min and max
values[1:3] // {2, 3, 4}

// Determining only max will use min = 0
values[:2] // {1, 2, 3}

// Determining only min will use max = last element
values[3:] // {3, 4}

// Length: number of elements that a slice contains
len(values) // 5

// Capacity: number of elements that a slice can contain
values = values[:1]
len(values) // 2
cap(values) // 5

// Slice literal
slice := []bool{true, true, false}

// make function: create a slice with length and capacity
slice := make([]int, 5, 6) // make(type, len, cap)
```

```
// Append new element to slice
slice := []int{ 1, 2 }
slice = append(slice, 3)
slice // { 1, 2, 3 }
slice = append(slice, 3, 2, 1)
slice // { 1, 2, 3, 3, 2, 1 }

// For range: iterate over a slice
slice := string["W", "o", "w"]

for i, value := range slice {
    i // 0, then 1, then 2
    value // "W", then "o", then "w"
}

// Skip index or value

for i := range slice {
    i // 0, then 1, then 2
}

for _, value := range slice {
   value // "W", then "o", then "w"
}
```

Return to Summary

Functions

```
// Functions acts as a scoped block of code
func sayHello() {
    // Hello World!
}
sayHello() // Hello World!

// Functions can take zero or more parameters, as so return zero or more parameters
func sum(x int, y int) int {
```

```
    return x + y
}
sum(3, 7) // 10

// Returned values can be named and be used inside the function
func doubleAndTriple(x int) (double, triple int) {
    double = x * 2
    triple = x * 3
    return
}
d, t := doubleAndTriple(5)
// d = 10
// t = 15

// Skipping one of the returned values
_, t := doubleAndTriple(3)
// t = 9

// Functions can defer commands. Deferred commands are
// ran in a stack order after the execution and
// returning of a function
var aux = 0

func switchValuesAndDouble(x, y int) {
    aux = x
    defer aux = 0 // cleaning variable to post use
    x = y * 2
    y = aux * 2
}

a, b = 2, 5
switchValuesAndDouble(2, 5)

// a = 10
// b = 4
// aux = 0

// Functions can be handled as values and be anonymous functions
func calc(fn func(int, int) int) int {
    return fn(2, 6)
```

```
}

func sum(x, y int) int {
    return x + y
}

func mult(x, y int) int {
    return x * y
}

calc(sum) // 8
calc(mult) // 12
calc(
    func(x, y int) int {
□□return x / y
    }
) // 3

// Function closures: a function that returns a function
// that remembers the original context
func calc() func(int) int {
    value := 0
    return func(x int) int {
        value += x
        return value
    }
}

calculator := calc()
calculator(3) // 3
calculator(45) // 48
calculator(12) // 60
```

Return to Summary

Structs

Structs are a way to arrange data in specific formats.

```
// Declaring a struct
type Person struct {
    Name string
    Age int
}

// Initializing
person := Person{"John", 34}
person.Name // "John"
person.Age // 34

person2 := Person{Age: 20}
person2.Name // ""
person2.Age // 20

person3 := Person{}
person3.Name // ""
person3.Age // 0
```

Return to Summary

Maps

Maps are data structures that holds values assigned to a key.

```
// Declaring a map
var cities map[string]string

// Initializing
cities = make(map[string]string)
cities // nil

// Insert
cities["NY"] = "EUA"

// Retrieve
newYork = cities["NY"]
newYork // "EUA"
```

```
// Delete
delete(cities, "NY")

// Check if a key is set
value, ok := cities["NY"]
ok // false
value // ""
```

Return to Summary

Pointers

Pointers are a direct reference to a memory address that some variable or value is being stored.

```
// Pointers has *T type
var value int
var pointer *int

// Point to a variable memory address with &
value = 3
pointer = &value

pointer // 3
pointer = 20
pointer // 20
pointer += 5
pointer // 25

// Pointers to structs can access the attributes
type Struct struct {
    X int
}

s := Struct{3}
pointer := &s

s.X // 3
```

Obs: Unlike C, Go doesn't have pointer arithmetics.

Return to Summary

Methods and Interfaces

Go doesn't have classes. But you can implement methods, interfaces and almost everything contained in OOP, but in what gophers call "Go Way"

```
type Dog struct {
    Name string
}

func (dog *Dog) bark() string {
    return dog.Name + " is barking!"
}

dog := Dog{"Rex"}
dog.bark() // Rex is barking!
```

Interfaces are implicitly implemented. You don't need to inform that your struct are correctly implementing a interface if it already has all methods with the same name of the interface. All structs implement the `interface{}` interface. This empty interface means the same as `any`.

```
// Car implements Vehicle interface
type Vehicle interface {
    Accelerate()
}

type Car struct {

}

func (car *Car) Accelerate() {
    return "Car is moving on ground"
}
```

Return to Summary

Errors

Go doesn't support `throw`, `try`, `catch` and other common error handling structures. Here, we use `error` package to build possible errors as a returning parameter in functions

```
import "errors"

// Function that contain a logic that can cause a possible exception flow
func firstLetter(text string) (string, error) {
    if len(text) < 1 {
        return nil, errors.New("Parameter text is empty")
    }
    return string(text[0]), nil
}

a, errorA := firstLetter("Wow")
a // "W"
errorA // nil

b, errorB := firstLetter("")
b // nil
errorB // Error("Parameter text is empty")
```

Return to Summary

Testing

Go has a built-in library to unit testing. In a separate file you insert tests for functionalities of a file and run `go test package` to run all tests of the actual package or `go test path` to run a specific test file.

```
// main.go
func Sum(x, y int) int {
```

```
    return x + y
}

// main_test.go
import (
    "testing"
    "reflect"
)

func TestSum(t *testing.T) {
    x, y := 2, 4
    expected := 2 + 4

    if !reflect.DeepEqual(Sum(x, y), expected) {
        t.Fatalf("Function Sum not working as expected")
    }
}
```

Return to Summary

Concurrency

One of the main parts that make Go attractive is its form to handle with concurrency. Different than parallelism, where tasks can be separated in many cores that the machine processor have, in concurrency we have routines that are more lightweight than threads and can run asynchronously, with memory sharing and in a single core.

```
// Consider a common function, but that function can delay itself because some processing
func show(from string) {
	for i := 0; i < 3; i++ {
		fmt.Printf("%s : %d\n", from, i)
	}
}

// In a blocking way...
func main() {
	show("blocking1")
```

```
	show("blocking2")

	fmt.Println("done")
}
/*  blocking1: 0
    blocking1: 1
    blocking1: 2
    blocking2: 0
    blocking2: 1
    blocking2: 2
    done
*/

// Go routines are a function (either declared previously or anonymous) called with the
keyword go
func main() {
	go show("routine1")
	go show("routine2")

	go func() {
		fmt.Println("going")
	}()

	time.Sleep(time.Second)

	fmt.Println("done")
}

/*  Obs: The result will depends of what processes first
    routine2: 0
    routine2: 1
    routine2: 2
    going
    routine1: 0
    routine1: 1
    routine1: 2
    done
*/

// Routines can share data with channels
```

```
// Channels are queues that store data between multiple routines
msgs := make(chan string)

go func(channel chan string) {
    channel <- "ping"
}(msgs)

go func(channel chan string) {
    channel <- "pong"
}(msgs)

fmt.Println(<-msgs) // pong
fmt.Println(<-msgs) // ping

// Channels can be bufferized. Buffered channels will accept a limited number of values and
when someone try to put belong their limit, it will throw and error
numbers := make(chan int, 2)

msgs<-0
msgs<-1
msgs<-2

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

// Channels can be passed as parameter where the routine can only send or receive
numbers := make(chan int)

go func(sender chan<- int) {
    sender <- 10
}(numbers)

go func(receiver <-chan int) {
    fmt.Println(<-receiver) // 10
}(numbers)

time.Sleep(time.Second)

// When working with multiple channels, the select can provide a control to execute code
accordingly of what channel has bring a message
c1 := make(chan string)
```

```
c2 := make(chan string)

select {
case msg1 := <-c1:
    fmt.Println("received", msg1)
case msg2 := <-c2:
    fmt.Println("received", msg2)
default:
    fmt.Println("no messages")
}

go func() {
    time.Sleep(1 * time.Second)
    c1 <- "channel1 : one"
}()
go func() {
    time.Sleep(2 * time.Second)
    c2 <- "channel2 : one"
}()

for i := 0; i < 2; i++ {
    select {
    case msg1 := <-c1:
        fmt.Println("received", msg1)
    case msg2 := <-c2:
        fmt.Println("received", msg2)
    }
}

/*
    no messages
    received channel1: one
    received channel2: one
*/

// Channels can be closed and iterated
channel := make(chan int, 5)

for i := 0; i < 5; i++ {
    channel <- i
```

```
}

close(channel)

for value := range channel {
    fmt.Println(value)
}

/*
    0
    1
    2
    3
    4
*/
```

Return to Summary

Package `fmt`

```
import "fmt"

fmt.Print("Hello World") // Print in console
fmt.Println("Hello World") // Print and add a new line in end
fmt.Printf("%s is %d years old", "John", 32) // Print with formatting
fmt.Errorf("User %d not found", 123) // Print a formatted error
```

Return to Summary

Revision #1
Created 2025-11-28 02:25:19 UTC by gasick
Updated 2025-11-28 02:25:31 UTC by gasick