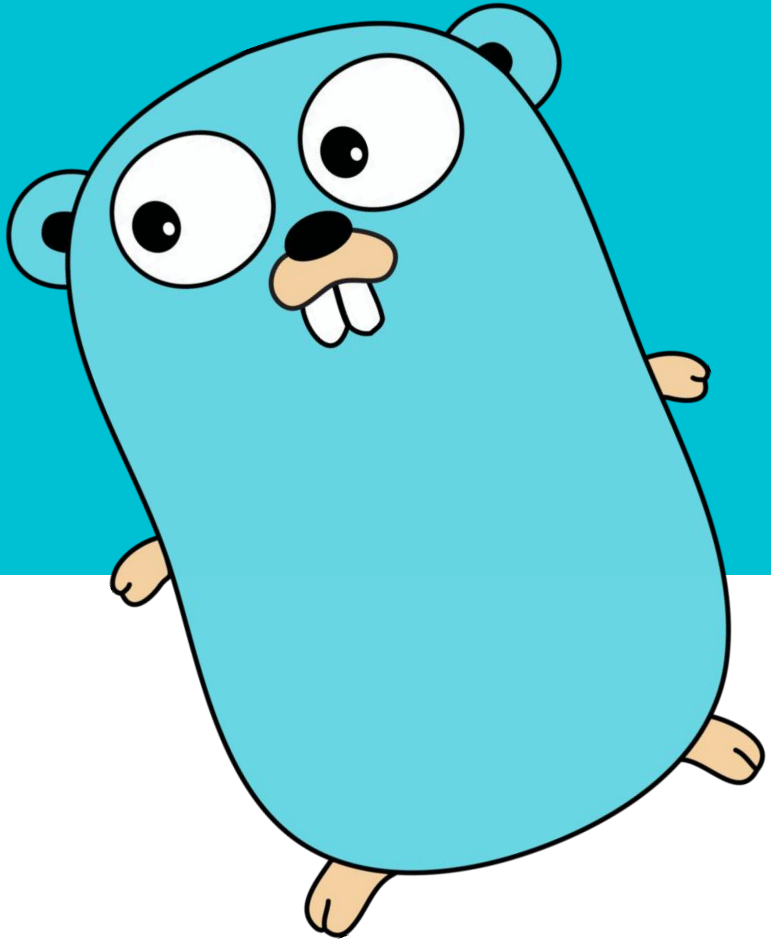
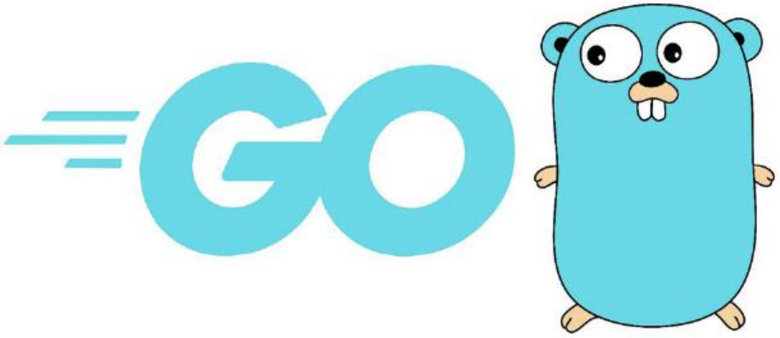
Golang first steps

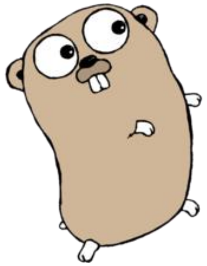


Laurent Guérin
Version 0.9 - 2019 October

Golang

Go, also known as **Golang**, is a statically typed, compiled programming language designed at **Google** by **Robert Griesemer**, **Rob Pike**, and **Ken Thompson**.



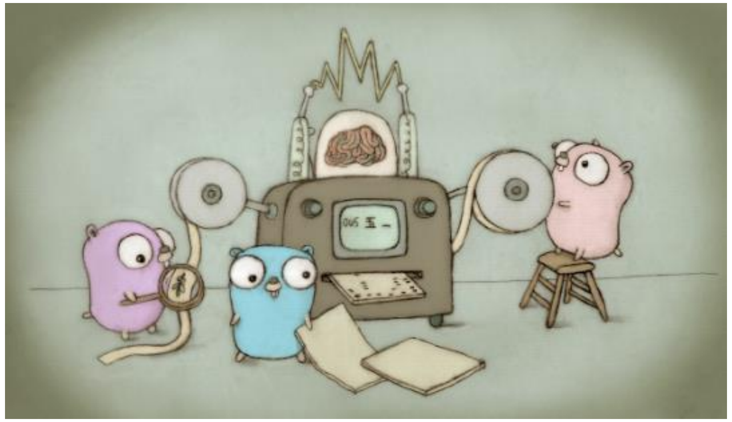


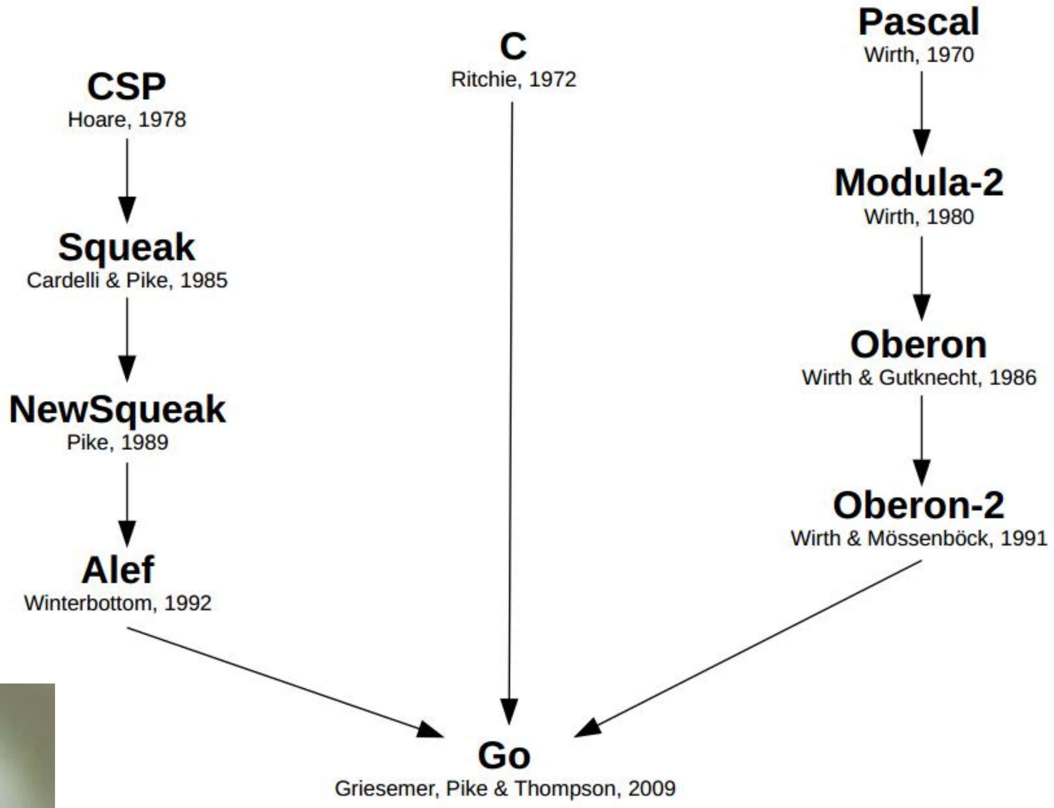
« Gopher » created by Renee French
Licence « Creative Commons »
https://golang.org/doc/gopher/

Golang – the history

"Go is an open source programming language that makes it easy to build simple, reliable, and efficient software."

https://golang.org/





CSP
Hoare, 1978

Squeak
Cardelli & Pike, 1985

NewSqueak
Pike, 1989

Alef
Winterbottom, 1992

C
Ritchie, 1972

Pascal
Wirth, 1970

Modula-2
Wirth, 1980

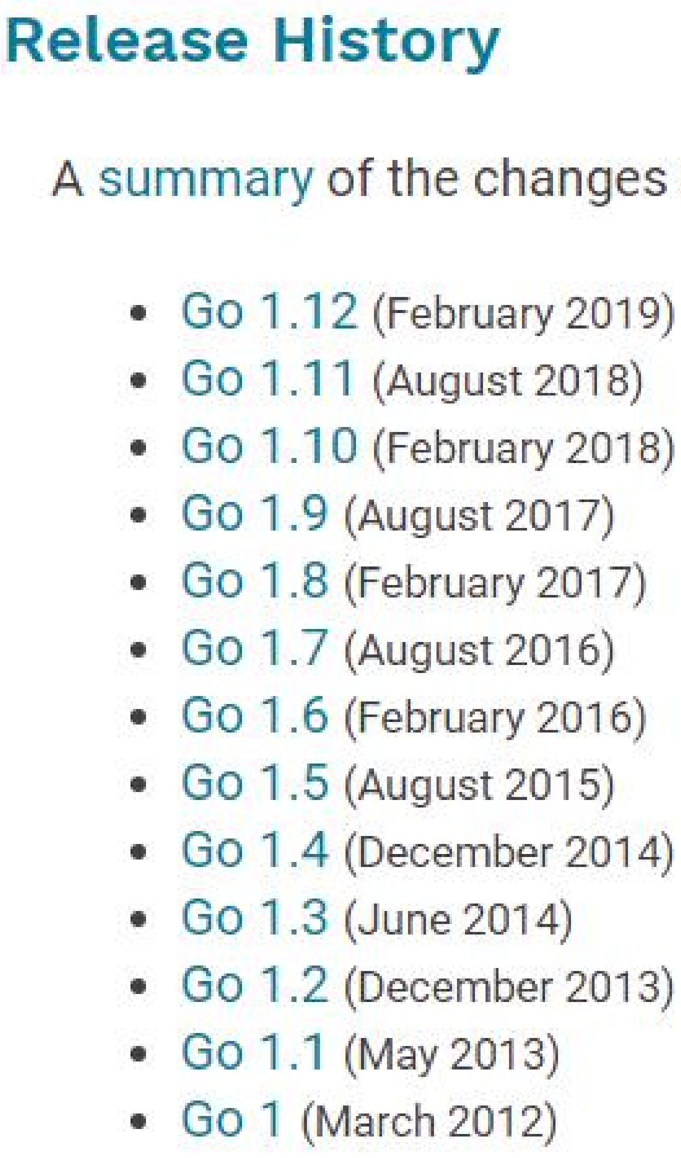
Oberon
Wirth & Gutknecht, 1986

Oberon-2
Wirth & Mössenböck, 1991

Go
Griesemer, Pike & Thompson, 2009

2009

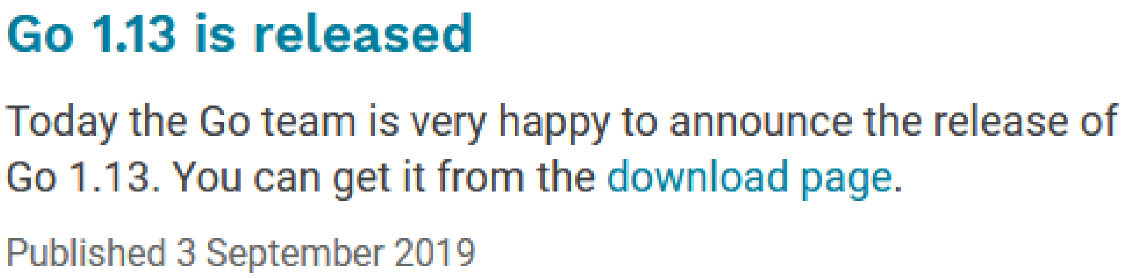
Golang – the history



Release History

A summary of the changes between Go releases. Notes for the major releases:

- Go 1.12 (February 2019)
- Go 1.11 (August 2018)
- Go 1.10 (February 2018)
- Go 1.9 (August 2017)
- Go 1.8 (February 2017)
- Go 1.7 (August 2016)
- Go 1.6 (February 2016)
- Go 1.5 (August 2015)
- Go 1.4 (December 2014)
- Go 1.3 (June 2014)
- Go 1.2 (December 2013)
- Go 1.1 (May 2013)
- Go 1 (March 2012)



Go 1.13 is released

Today the Go team is very happy to announce the release of Go 1.13. You can get it from the download page.

Published 3 September 2019

Golang – about Go …

Work started : 2007

Initial release : 2009

Started as “system language”

Like “C” … but simpler, cleaner and more concise

Only ~ 25 keywords

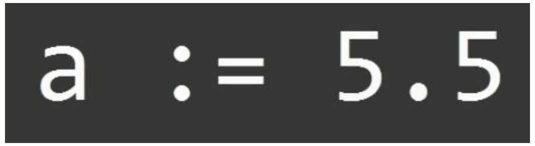
Strongly typed

Garbage collector

Compiled => no runtime & fast !

Cross platform

Type inference :



```
a := 5.5
```

« 5.5 » → ok, that’s a float !

Performances close to C language !

Golang - what's in Go ?

What you will **not** find in Go :

- ~~Inheritance~~
- ~~Classes~~
- ~~Constructor~~
- ~~Exceptions~~
- ~~Annotations~~
- ~~Generics~~

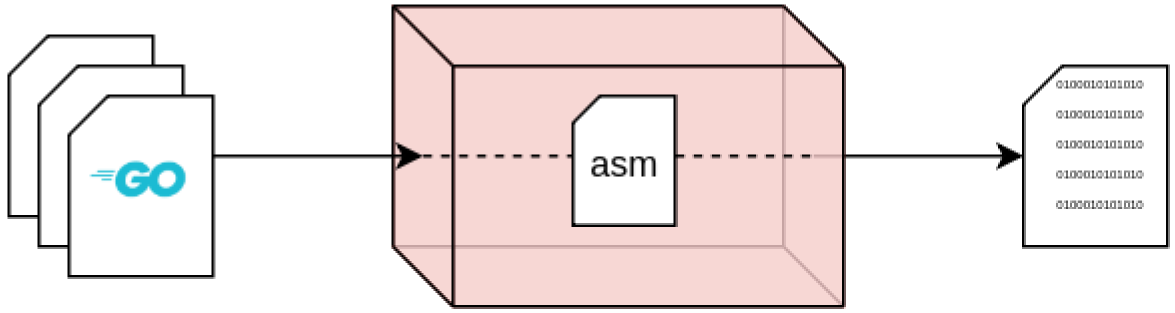
Go is not Object-Oriented !

What you will find in Go :

- Packages
- Interfaces
- Garbage collection
- Structures
- Functions (“first class citizen”)
- Concurrency (made easy)

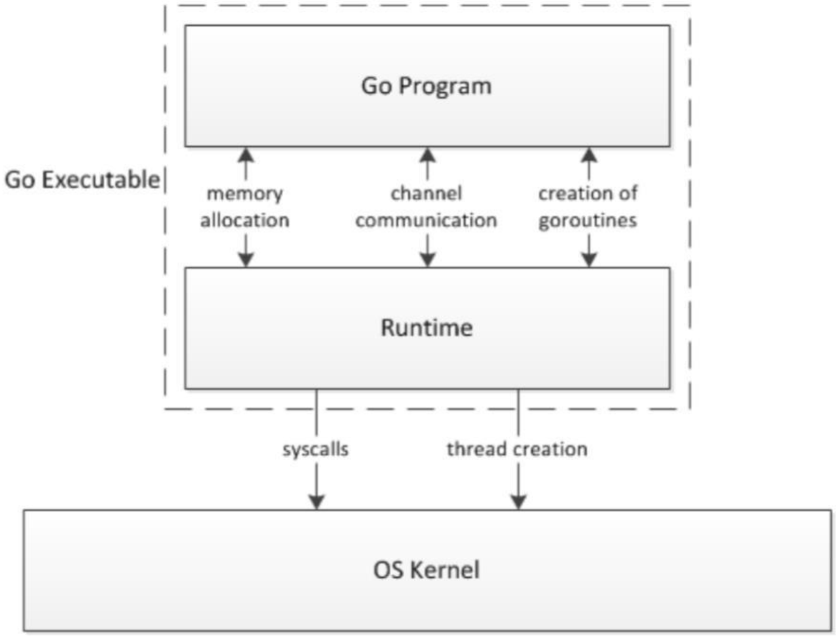
Golang - compilation

```
> go build myfolder
> go install myfolder
```



(Go source) (Compiler) (Executable Binary)

Executable binary
=> Nothing else to install / deploy
(no JRE / JVM, no interpreter, …)

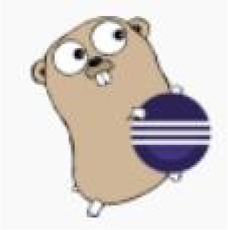


Cross compiler
=> you can cross compile for another architecture
or operating system on your local system
with 2 environment variables : **$GOOS** and **$GOARCH**

```
GOOS=linux   GOARCH=amd64  go build main.go
GOOS=windows GOARCH=amd64  go build main.go
```

Golang - which editor / IDE ?

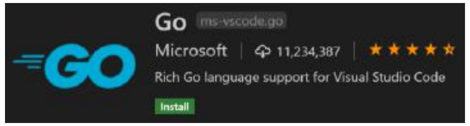
LiteIDE (designed for Go)
http://liteide.org/



Eclipse
with “**Goclipse**” plugin

GoLand
(JetBrains)

Atom
with “**go-plus**”



Go ms-vscode.go
Microsoft | 11,234,387 | ★★★★★
Rich Go language support for Visual Studio Code
Install

Visual Studio Code
with “**Go extension**”

vim

Sublime Text

Golang – Use cases

Backends for any type of **API** : REST, GraphQL or gRPC
(Dropbox, Uber and GitHub have built API's in Go)

Command Line Tools

Any software that interacts with the **OS** through its public API (Containerisation, Docker, Kubernetes, etc)

Server-side services : pub/sub, caching, high-CPU jobs, etc

Scalable or embedded **databases** (InfluxDB, Bolt, Dgraph, etc)

Cloud tooling

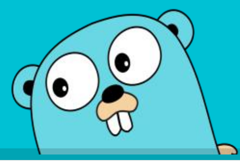
Scripting for DevOps (faster than Python)

WebAssembly / WASM (emerging)

Performance => cost reduction !

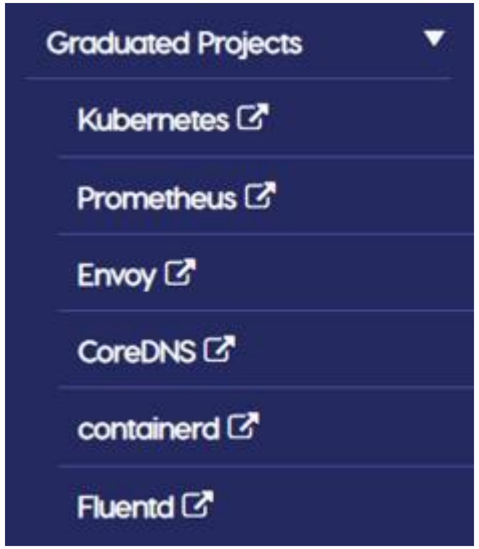
Cf Iron blog :
"How We Went from 30 Servers to 2"
(Ruby On Rails → Go)
https://blog.iron.io/how-we-went-from-30-servers-to-2-go/

Some projects written in Go



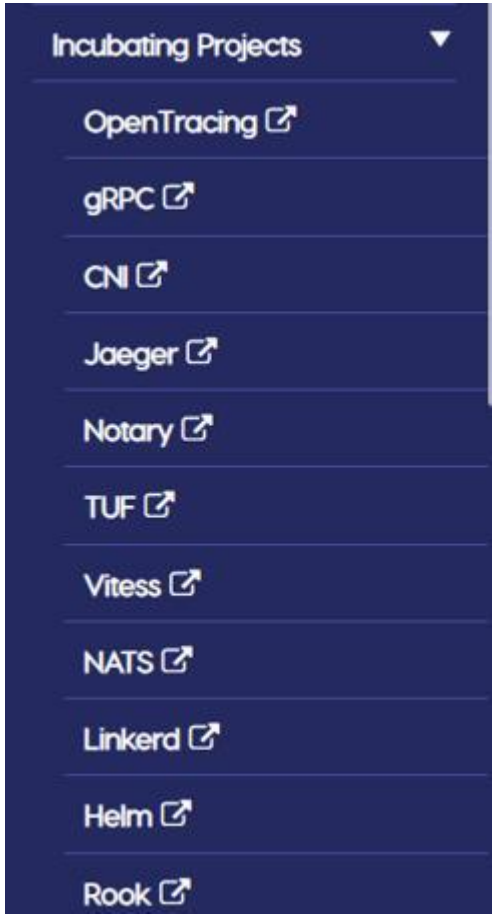
CNCF projects : https://www.cncf.io/projects/

CLOUD NATIVE
COMPUTING FOUNDATION



Graduated Projects ▼

- Kubernetes
- Prometheus
- Envoy
- CoreDNS
- containerd
- Fluentd



Incubating Projects ▼

- OpenTracing
- gRPC
- CNI
- Jaeger
- Notary
- TUF
- Vitess
- NATS
- Linkerd
- Helm
- Rook

Kubernetes
Prometheus
Helm
gRPC

docker

Caddy server (https://caddyserver.com/)

Traefik (https://traefik.io/)

InfluxDB (https://www.influxdata.com/)

Hugo (https://gohugo.io/)

Grafana (https://grafana.com/)

Hashicorp tools :
(https://www.hashicorp.com/)

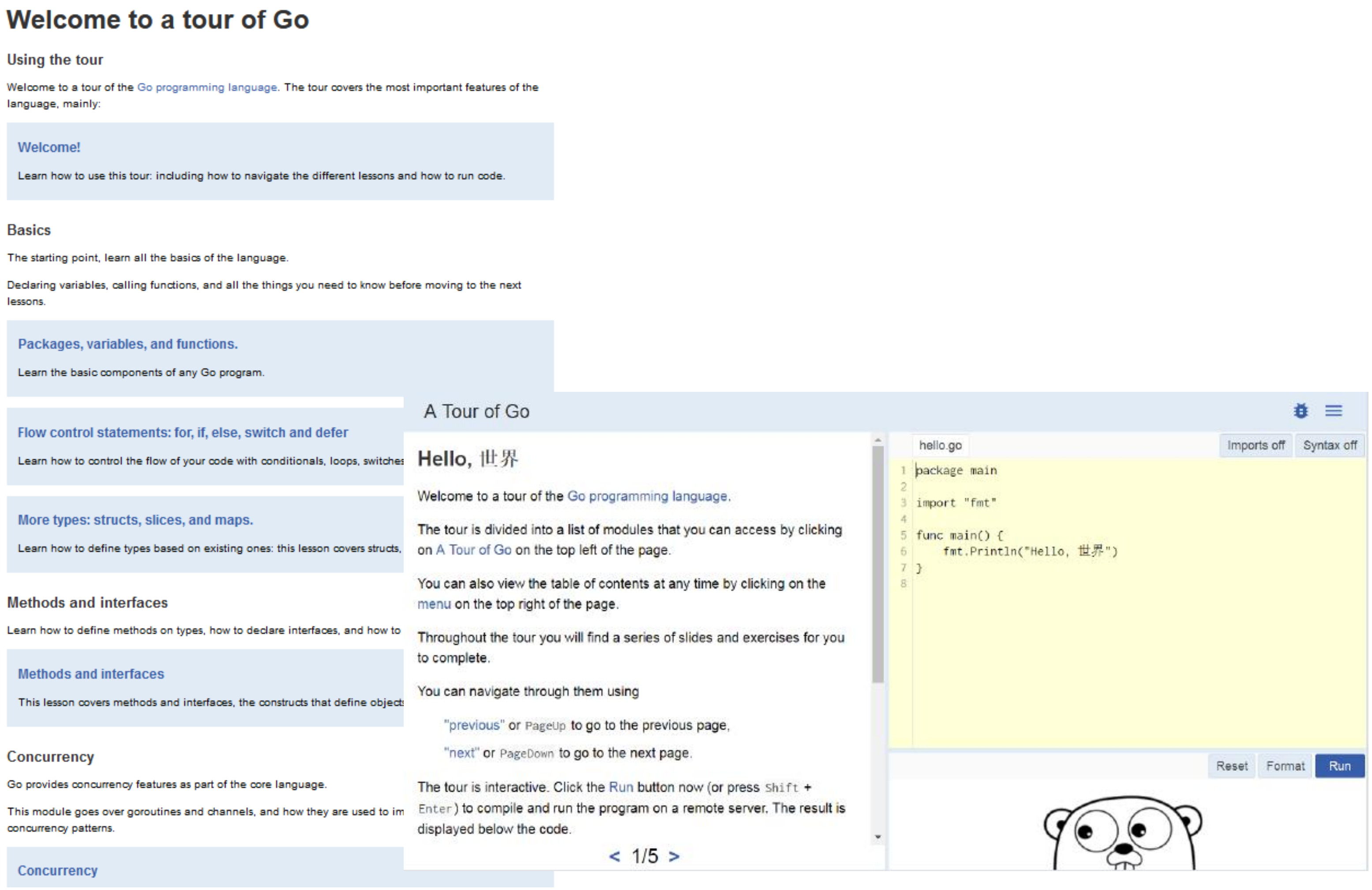
- **Consul**
- **Nomad**
- **Terraform**
- etc

Gitea (https://gitea.io/)
Gogs (https://gogs.io/)

Flynn (PaaS) (https://flynn.io/)

Golang – Main pointers

- Official Web Site : https://golang.org/
- Documentation : https://golang.org/doc/
- A Tour of Go : https://tour.golang.org/list https://tour.golang.org



Welcome to a tour of Go

Using the tour

Welcome to a tour of the Go programming language. The tour covers the most important features of the language, mainly:

Welcome!

Learn how to use this tour: including how to navigate the different lessons and how to run code.

Basics

The starting point, learn all the basics of the language.

Declaring variables, calling functions, and all the things you need to know before moving to the next lessons.

Packages, variables, and functions.

Learn the basic components of any Go program.

Flow control statements: for, if, else, switch and defer

Learn how to control the flow of your code with conditionals, loops, switches

More types: structs, slices, and maps.

Learn how to define types based on existing ones: this lesson covers structs,

Methods and interfaces

Learn how to define methods on types, how to declare interfaces, and how to

Methods and interfaces

This lesson covers methods and interfaces, the constructs that define objects

Concurrency

Go provides concurrency features as part of the core language.

This module goes over goroutines and channels, and how they are used to im concurrency patterns.

Concurrency

A Tour of Go

Hello, 世界

Welcome to a tour of the Go programming language.

The tour is divided into a list of modules that you can access by clicking on A Tour of Go on the top left of the page.

You can also view the table of contents at any time by clicking on the menu on the top right of the page.

Throughout the tour you will find a series of slides and exercises for you to complete.

You can navigate through them using

"previous" or `PageUp` to go to the previous page,

"next" or `PageDown` to go to the next page.

The tour is interactive. Click the Run button now (or press `Shift + Enter`) to compile and run the program on a remote server. The result is displayed below the code.

< 1/5 >

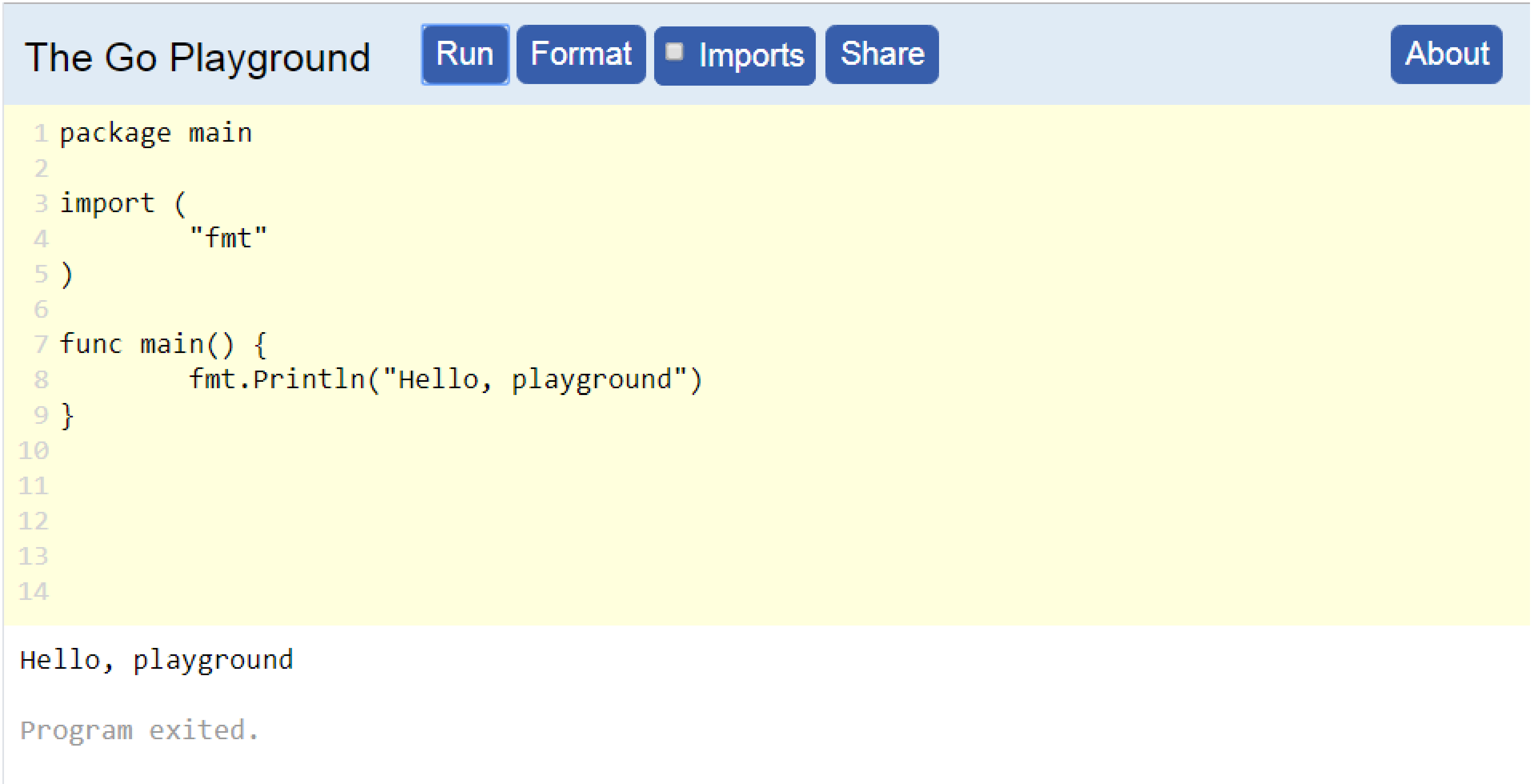
hello.go

```go
package main

import "fmt"

func main() {
	fmt.Println("Hello, 世界")
}
```

https://github.com/avelino/awesome-go

Golang - let's try it online...

- The Go Playground : https://play.golang.org



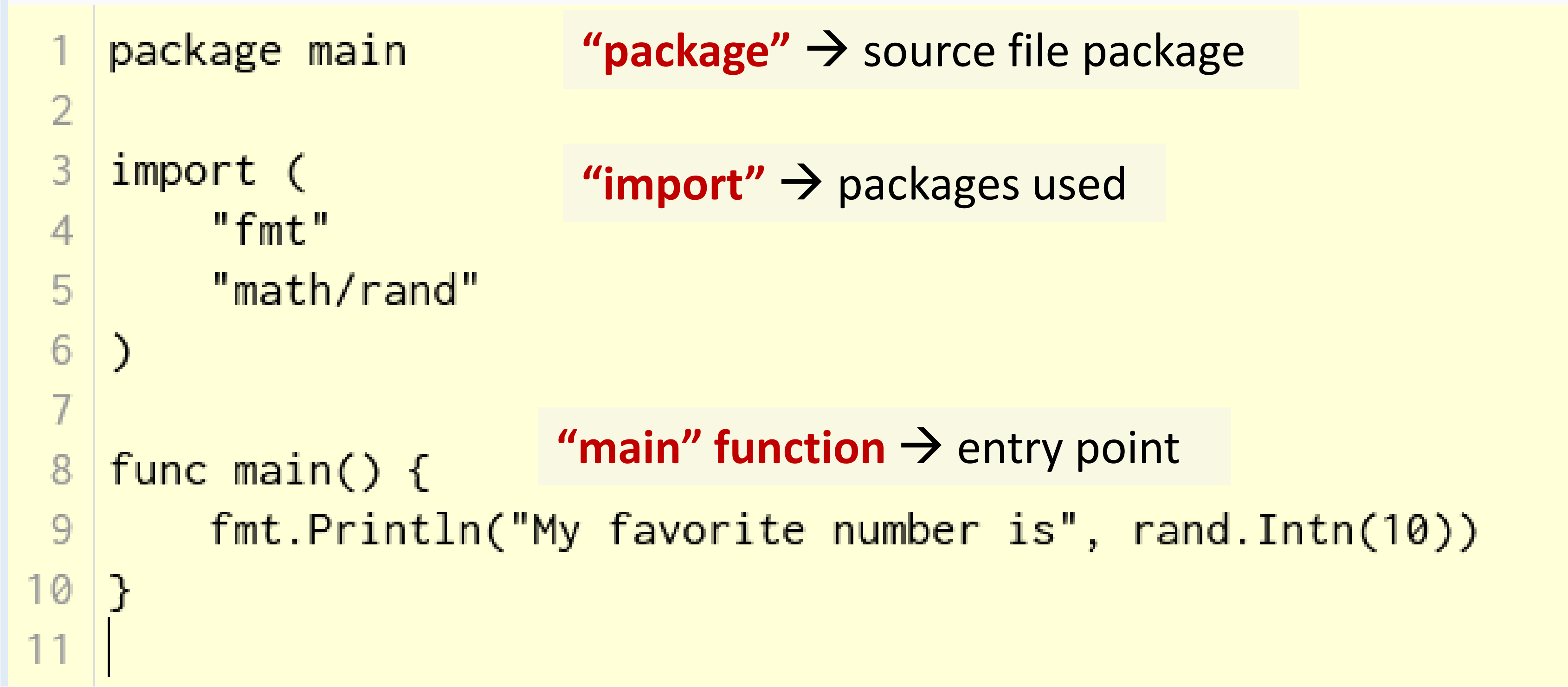
```go
package main

import (
	"fmt"
)

func main() {
	fmt.Println("Hello, playground")
}
```

```
Hello, playground

Program exited.
```

Packages

Golang – Packages

Every Go program is made up of **packages**
A program starts running in package **“main”**



```go
package main

import (
	"fmt"
	"math/rand"
)

func main() {
	fmt.Println("My favorite number is", rand.Intn(10))
}

```

“package” → source file package

“import” → packages used

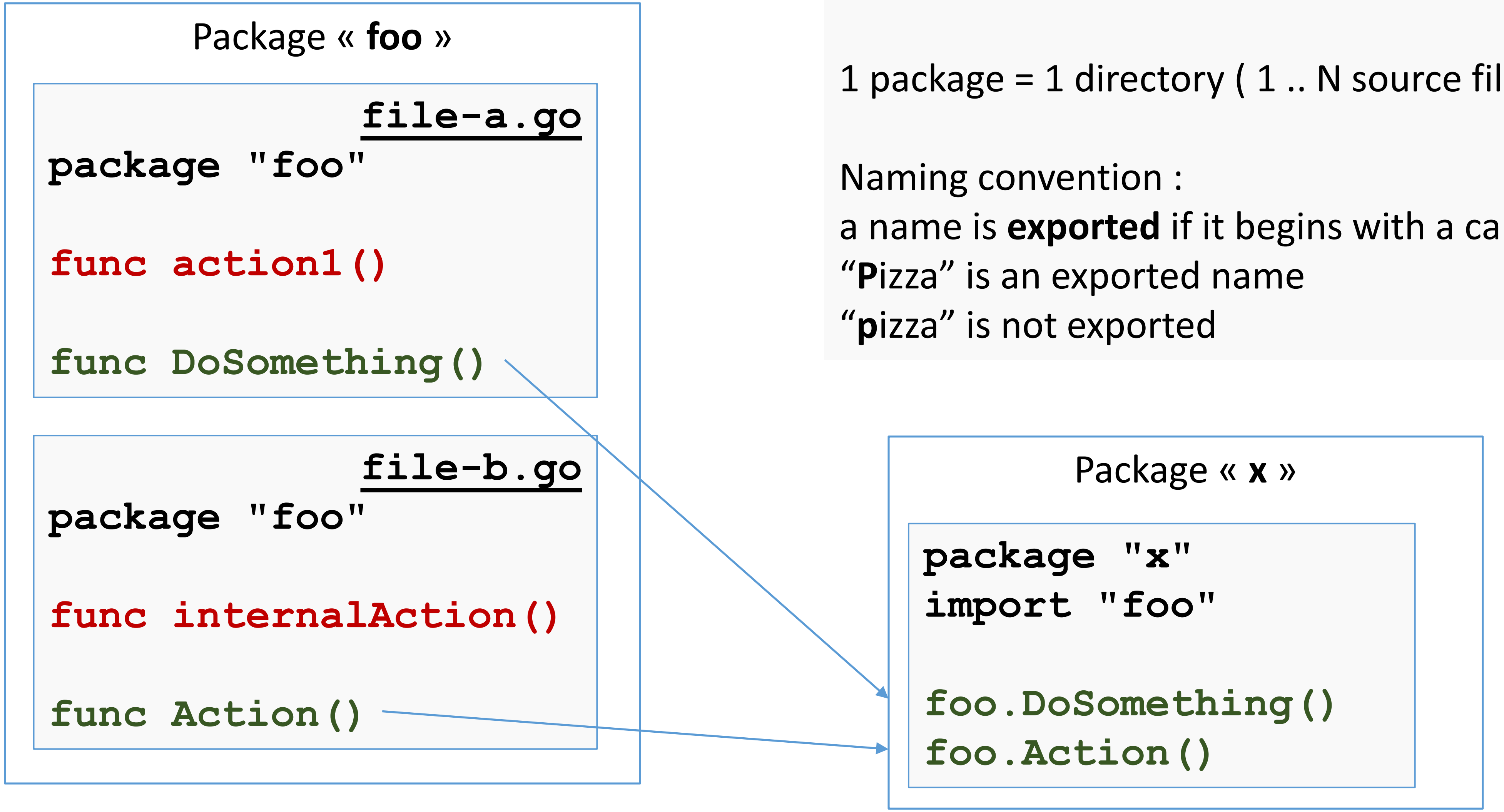
“main” function → entry point

By convention, the package name is the same as the last element of the import path.
Path : “math/rand”
Name → “rand”

`fmt.Println(..)` → call “**`Println`**” function defined in package “**`fmt`**”
`rand.Intn(..)` → call “**`Intn`**” function defined in package “**`rand`**”

Golang – Packages

/home/user/go/src/github.com/myproject/aaa/bbb/foo



Package « **foo** »

file-a.go

```
package "foo"

func action1()

func DoSomething()
```

file-b.go

```
package "foo"

func internalAction()

func Action()
```

Package « **x** »

```
package "x"
import "foo"

foo.DoSomething()
foo.Action()
```

Conventions :

1 package = 1 directory (1 .. N source files)

Naming convention :
a name is **exported** if it begins with a capital letter.
“**P**izza” is an exported name
“**p**izza” is not exported

Variables, Types & Constants

Built-in functions

Control flow (if…else, for, switch)

Golang – Variables

```
// variable declaration (value = 0)
var age int

age = 29 // variable assignment
fmt.Println("age is ", age) // usage
```

Type is AFTER variable name

Variable :

- **name** « age »
- **type** « int »
- **initial value**

```
// variable declaration with initial value
var age int = 29
```

NO “ ; “

```
// variable declaration with type inference -> int
age := 29
```

Type inference with “ := “

```
// declaring multiple variables
var width, height int
var width, height int = 100, 50
```

Golang – Variables

```
var (
      name   = "bob"
      age    = 12
      height int
    )

name, age := "bob", 12 // short hand declaration
age = "x" // error type is int, not string

a, b := 20, 30 // declare variables a and b
b, c := 40, 50 // b is already declared but c is new
a, b := 40, 50 // error, no new variables

b, c = 80, 90 // assign new values
```

NB : if variable declared but not used => compilation error !
(error : x declared and not used)

Golang – Types

Boolean :

- bool

String :

- string

String → UTF-8

```
a := true
b := false
c := a && b
d := a || b
```

```
first := "Bob"
last := "Sponge"
age := 12
name := first + " " + last
name = name + " : " + age // error
           ( mismatched types string and int )
name = name + " : " + strconv.Itoa(age)// OK
fmt.Println(name)
```

Golang – Types

Numeric :

- int8, int16, int32, int64, int
- uint8, uint16, uint32, uint64, uint
- float32, float64

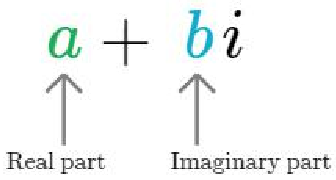
- complex64, complex128

- *byte* (byte is an alias of uint8)
- *rune* (rune is an alias of int32)

```
x := 10
var y float64 = x // error
    cannot use x (type int) as type float64
var z float64 = float64(x) // OK
```

```
i := 55       // int
j := 67.8     // float64
sum := i + j // error
    invalid operation (mismatched types)
sum := i + int(j) // OK, j is converted to int
```

```
// complex numbers
c1 := complex(5, 7)
c2 := 6 + 7i
```



Golang – Constants

Keyword « **const** »

```
const a = 55 //allowed
a = 89 // error : reassignment not allowed
```

The value of a constant must be known at compile time.

```
var   a = math.Sqrt(4) // allowed
const b = math.Sqrt(4) // error : not allowed
                          math.Sqrt(4) is not a constant
```

Golang – Built-in functions

Built-in functions are **predeclared**.
They can only appear in call expressions (they cannot be used as function values).

// Creations

- func **new**(Type) *Type
- func **make**(t Type, size ...IntegerType) Type
  (make for *slices*, *maps* and *channels*)

// Length and capacity

- func **len**(v Type) int
- func **cap**(v Type) int

// Append, delete, copy

- func **append**(slice []Type, elems ...Type) []Type
- func **delete**(m map[Type]Type1, key Type)
- func **copy**(dst, src []Type) int

// Complex types

- func **real**(c ComplexType) FloatType
- func **complex**(r, i FloatType) ComplexType
- func **imag**(c ComplexType) FloatType

// Errors handling

- func **panic**(v interface{})
- func **recover**() interface{}

// Channels

- func **close**(c chan<- Type)

// Print on standard error

- func **print**(args ...Type)
- func **println**(args ...Type)

Golang – Control Structures – “if / else”

```
if condition {

}
```

OK

```
if num % 2 == 0 {
    fmt.Println("even")
} else {
    fmt.Println("odd")
}
```

syntax error

```
if num % 2 == 0 {
    fmt.Println("even")
} ( new line )
else {
    fmt.Println("odd")
}
```

NB :

- **formatting is imposed !**
  (no new line anywhere)
- **parentheses not required**

```
num := 99
if num <= 50 {
    fmt.Println("less than or equal to 50")
} else if num >= 51 && num <= 100 {
    fmt.Println("between 51 and 100")
} else {
    fmt.Println("greater than 100")
}
```

```
if statement; condition {

}
```

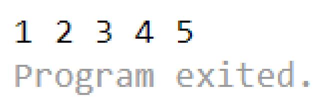
```
if num := 10; num % 2 == 0 {
    fmt.Println("even")
}  else {
    fmt.Println("odd")
}
```

Golang – Control Structures – “for”

```
for initialization; condition; post {

}
```

```
for i := 1; i <= 10; i++ {
    fmt.Printf(" %d",i)
}
```

```
for i := 1; i <= 10; i++ {
     if i > 5 {
          break //loop is terminated
     }
     fmt.Printf("%d ", i)
}
```

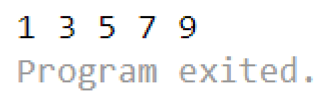


```
1 2 3 4 5
Program exited.
```

```
// Loop forever
for {
}
```

```
i := 2
for ; i <= 10; {
    fmt.Printf(" %d",i)
    i += 2
}
```

```
for i := 1; i <= 10; i++ {
     if i%2 == 0 {
          continue
     }
     fmt.Printf("%d ", i)
}
```

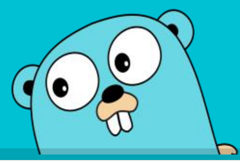


```
1 3 5 7 9
Program exited.
```

Nested loop

```
for i := 0; i < 3; i++ {
    for j := 1; j < 4; j++ {
        fmt.Printf(" %d , %d\n", i, j)
        if i == j {
            break
        }
    }
}
```

Golang – Control Structures – “switch”



```
switch xxxx {

}
```

```
x := 4
switch x {
case 1:
   fmt.Println("One")
case 2:
   fmt.Println("Two")
case 3:
   fmt.Println("Three")
case 4:
   fmt.Println("Four")
case 5,6,7: // multiple
   fmt.Println("Five/Six/Seven")
default: //default case
   fmt.Println("Other")
}
```

Error : "duplicate case 4 in switch"

```
x := 4
switch x {
case 1:
       fmt.Println("One")
case 4:
       fmt.Println("Four")
case 4:
       fmt.Println("Five")
}
```

```
num := 12
switch { // expression is omitted
   case num >= 0 && num <= 100:
      fmt.Println(">= 0 and <= 100")
   case num >= 101:
      fmt.Println(">= 100")
}
```

Functions

Pointers

Structures

Methods

Golang – Functions – “func”

```
func functionName(param_name type) returntype {
 //function body
}
```

NB :
Parameter type after name
Return type at the end

```
func print(msg string) {
        fmt.Println(msg)
}

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

func main() {
        print("foo")
        r := add(2, 3)
        fmt.Println(r)
}
```

Error : "missing function body"

```
func add(x int, y int) int
{
   return x + y
}
```

(new line)

Golang – Functions – “func”

It is possible to return multiple values from a function

```go
// Returns 2 'int'
func move(x int, y int) (int, int) {
   x2 := x + 100
   y2 := y + 100
   return x2, y2
}

func main() {
   x,y := move(30,70)
   fmt.Println(x,y)
}
```

Named return values

```go
// Returns 2 'int'
func move(x int, y int) (x2, y2 int) {
   x2 = x + 100   // x2 already declared
   y2 = y + 100   // y2 already declared
   return     // return x2, y2
}

func main() {
   x,y := move(30,70)
   fmt.Println(x,y)
}
```

Golang – “Blank Identifier”

Reminder : all declared variable must be used

```
// Returns 2 'int'
func myfunction(x int) (int, int) {
   ...
}

func main() {
    x,y := myfunction(30)
    use(y)
    // don’t want to use ‘x’
}
```

Error : "x declared and not used"

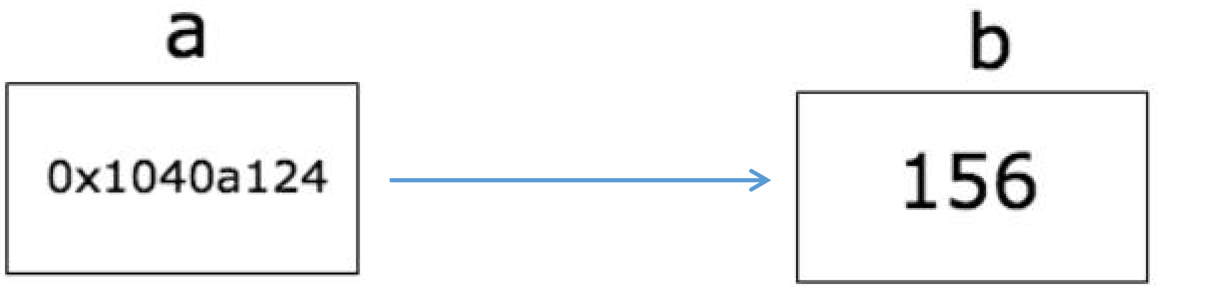
Solution : use the “blank identifier" "_" (underscore)

```
func main() {
    _, y := myfunction(30)
    use(y)
}
```

```
func main() {
    y := 123
    _ = 12
    _ = y
}
```

Golang – Pointers

- A pointer is a variable which stores the memory address of another variable.



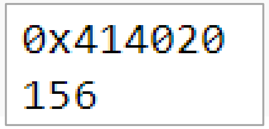
a holds the address of **b** **(“a points to b”)**

"*" + type = pointer for this type

"&" + var = address of this var

"*" + var = content of this var (“dereferencing”)

```
func main() {
  b := 156
  var a *int // type “int pointer”
  a = &b // address of b
  fmt.Println(a)
  fmt.Println(*a)
}
```



```
0x414020
156
```

```
func main() {
  b := 156 // int
  a := &b // address of b, type *int
  fmt.Println(a)
  fmt.Printf("Type of a : %T\n", a)
}
```

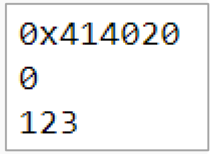
Golang – Pointers

The “zero value” of a pointer is “nil”

```
func main() {
  b := 25
  var a *int
  if a == nil {
    fmt.Println("a is", a)
    a = &b
    fmt.Println("a is", a)
  }
}
```

Pointer creation with ‘new’

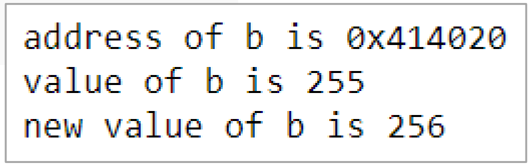
```
func main() {
  x := new(int) // *x = 0
  fmt.Println(x)
  fmt.Println(*x)
  *x = 123
  fmt.Println(*x)
}
```



```
0x414020
0
123
```

Change the “pointed” variable

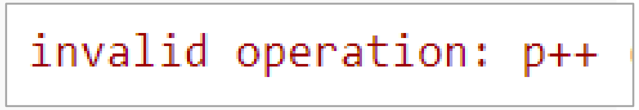
```
func main() {
  b := 255
  a := &b
  fmt.Println("address of b is", a)
  fmt.Println("value of b is", *a)
  *a++ // increment b !
  fmt.Println("new value of b is", b)
}
```



```
address of b is 0x414020
value of b is 255
new value of b is 256
```

pointer++ not supported

```
func main() {
  s := "abc"
  p := &s
  fmt.Println(" p = ",  p)
  fmt.Println("*p = ", *p)
  p++
}
```



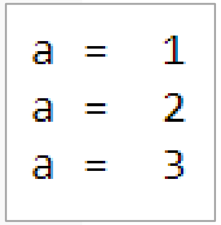
```
invalid operation: p++
```

Go is not
‘C language’

Golang – Pointers & functions

Pointer as function parameter

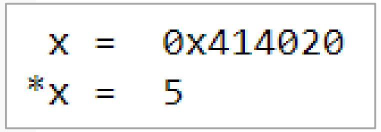
```go
func increment(val *int) {
     *val = *val + 1
}

func main() {
     a := 1
     fmt.Println("a = ",a)
     b := &a
     increment(b)
     fmt.Println("a = ",a)
     increment(&a)
     fmt.Println("a = ",a)
}
```



```
a =  1
a =  2
a =  3
```

Returning a pointer

```go
func getIntPtr() *int {
     i := 5
     return &i
}

func main() {
     x := getIntPtr()
     fmt.Println(" x = ",  x)
     fmt.Println("*x = ", *x)
}
```



```
 x =  0x414020
*x =  5
```

Golang - Structures

Structures are the way to create user-defined concrete types

```go
type Person struct {
     firstName string
     lastName  string
     age       int
}
```

```go
type Person struct {
     firstName, lastName string
     age, weight     int
}
```

"**named structure**" = a **new type** named "**Person**“
(the name can start with a lowercase)

```go
var person struct {
     firstName string
     lastName  string
     age       int
}
```

"**anonymous structure**"
(no new type)
(can be useful if not reusable)

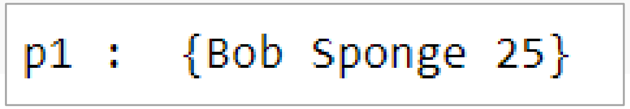
Golang – Named structure creation

Creation with field names :

```
p1  := Person {
             firstName:  "Bob",
             lastName:   "Sponge",
             age:        25,
      }
```

(trailing comma is not a typo)

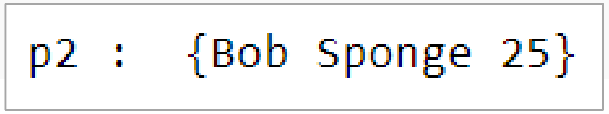
```
fmt.Println("p1  :  ", p1)
```



```
p1  :   {Bob Sponge 25}
```

Creation without field names :

```
p2  := Person {"Bob",
             "Sponge", 25 }

fmt.Println("p2  :  ", p2)
```



```
p2  :   {Bob Sponge 25}
```

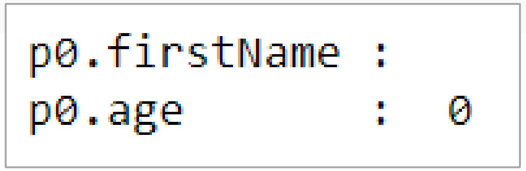
Comparison :

```
if ( p1 == p2 ) {
  ..
}
```

(equals if all fields are equal)

“Zero valued” structure
(not explicitly initialized)

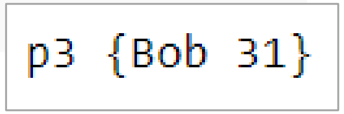
```
var p0 Person // no value

fmt.Println("p0.firstName : ",
                p0.firstName )
fmt.Println("p0.age       : ",
                p0.age       )
```



```
p0.firstName :
p0.age       :  0
```

Golang – Anonymous structure creation

```
p3 := struct {
        name string
        age int
} {
        name:  "Bob",
        age:   31,
}

fmt.Println("p3", p3)
```



```
p3 {Bob 31}
```

Structure definition (no ‘type’)

+

Fields values

Creation without field names :

```
p3:= struct {
            name string
            age int
      } {"Bob", 31 }

fmt.Println("p3", p3)
```

Golang – Nested structure

Structure :

```
type Address struct {
    city, state string
}

type Person struct {
    name string
    age int
    address Address
}
```

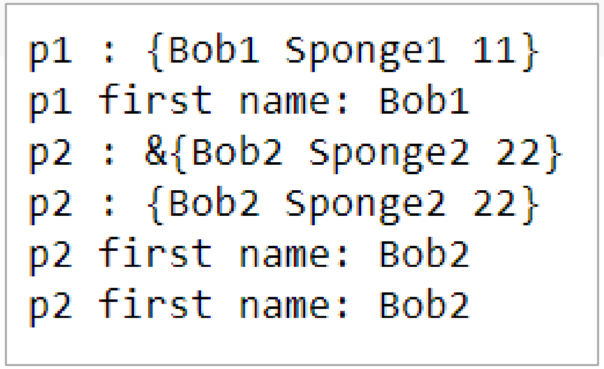
Usage :

```
var p Person
p.name = "Bob"
p.age = 50
p.address = Address{
      city:  "Boston",
      state: "Massachusetts",
}
fmt.Println("Name:", p.name)
fmt.Println("Age:", p.age)
fmt.Println("City:", p.address.city)
fmt.Println("State:", p.address.state)
```

Golang – Structures and pointers

```
p1 := Person  {"Bob1", "Sponge1", 11 }

fmt.Println("p1 :", p1)
fmt.Println("p1 first name:", p1.firstName)



p2 := &Person {"Bob2", "Sponge2", 22 } // Pointer

fmt.Println("p2 :", p2)
fmt.Println("p2 :", *p2)
fmt.Println("p2 first name:", (*p2).firstName)
fmt.Println("p2 first name:", p2.firstName) //same as *p2
```

p2 instead of ***p2**
→ no error



```
p1 : {Bob1 Sponge1 11}
p1 first name: Bob1
p2 : &{Bob2 Sponge2 22}
p2 : {Bob2 Sponge2 22}
p2 first name: Bob2
p2 first name: Bob2
```

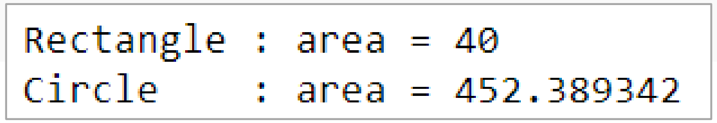
Golang – Methods

- Go does not have classes, but we can define **methods** on **types**.
- A method is nothing but a function with a special **receiver** argument.

```
func (receiver Type) methodName(parameter list) (returnTypes) {
}
```

```
type Rectangle struct {
        length int
        width  int
}
func (r Rectangle) Area() int {
        return r.length * r.width
}

type Circle struct {
        radius float64
}
func (c Circle) Area() float64 {
        return math.Pi * c.radius * c.radius
}
```

```
func main() {

  r := Rectangle{10, 4}
  fmt.Printf("Rectangle : area = %d\n",
      r.Area())

  c := Circle{12}
  fmt.Printf("Circle    : area = %f",
      c.Area())
}
```



```
Rectangle : area = 40
Circle    : area = 452.389342
```

Golang – Method receiver : value or pointer ?

- **Methods** with **value receivers**
  will accept both pointer and value receiver

```
func (r Rectangle) Area() int {
    return r.length * r.width
}
```

- **Methods** with **pointer receivers**
  will accept both pointer and value receiver

```
func (r *Rectangle) Area() int {
    return r.length * r.width
}
```

Pointer receiver is required to change struct fields

```
r.length = v
```

Calling method with value and pointer :

```
func main() {

  r1 := Rectangle{10, 4}
  fmt.Printf("Rectangle : area = %d\n",
      r1.Area())  // OK


  r2 := &Rectangle{10, 4} // Pointer
  fmt.Printf("Circle    : area = %d\n",
      r2.Area())  // OK
}
```

it's possible to ..

→ call ‘value receiver’ with a ‘pointer’

→ call ‘pointer receiver’ with a ‘value’

whereas **functions** with pointer **arguments** will accept only pointers and **functions** with value **arguments** will accept only values

Collections

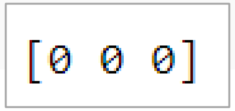
- **Array**
- **Slice**
- **Map**

(+ range, make, len, cap, append, delete)

Golang – Array

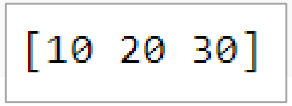
- **Array** = collection of elements that belong to the same type.
- All elements in an array are automatically assigned the zero value of the array type

```go
//int array with length 3
var a [3]int
fmt.Println(a)
```



```
[0 0 0]
```

```go
//int array with length 3
var a [3]int
a[0] = 10 // index starts at 0
a[1] = 20
a[2] = 30
fmt.Println(a)
```



```
[10 20 30]
```

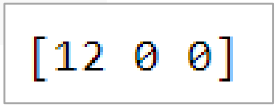
Short hand declaration :

```go
a := [3]int {10, 20, 30}
```

```go
a[3] = 40 // error
```

```
invalid array index 3 (out of bounds for 3-element array)
```

```go
a := [3]int {12}
fmt.Println(a)
```



```
[12 0 0]
```

Let the compiler determine the length :

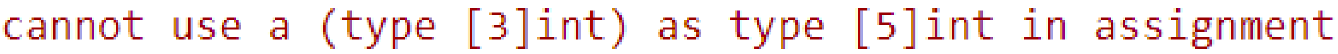
```go
a := [...]int {10, 20, 30, 40}
```

```go
fmt.Println("length : ", len(a))
```

Golang – Array

- The **size** of the array is a **part of the type**
  => [3]int and [5]int are distinct types

```
a := [3]int{5, 78, 8}
var b [5]int
b = a  // error
```

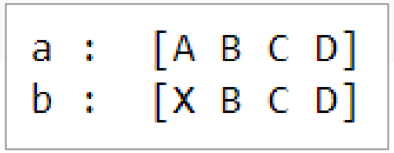


```
cannot use a (type [3]int) as type [5]int in assignment
```

- Hence arrays **cannot be resized** !
  (possible with “slices”)

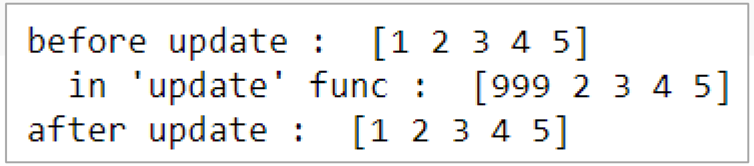
- Arrays are **value types** (not reference type)

```
a := [...]string{"A", "B", "C", "D" }
b := a // a copy of a is assigned to b
b[0] = "X"
fmt.Println("a : ", a)
fmt.Println("b : ", b)
```



```
a :  [A B C D]
b :  [X B C D]
```

```
func update(num [5]int) {
    num[0] = 999
    fmt.Println("  in 'update' func : ", num)
}
func main() {
    num := [...]int{1, 2, 3, 4, 5}
    fmt.Println("before update : ", num)
    update(num) // passed by value
    fmt.Println("after update : ", num)
}
```



```
before update :  [1 2 3 4 5]
  in 'update' func :  [999 2 3 4 5]
after update :  [1 2 3 4 5]
```

Without effect !

Golang – Array

- Iterating arrays (classical form)

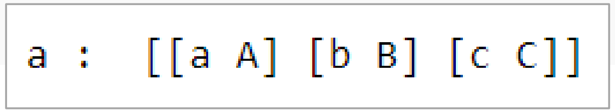
```
a := [...]float64{1.1, 2, 3.0, 4}
for i := 0; i < len(a); i++ {
   fmt.Printf("position %d : value = %.2f\n", i, a[i])
}
```

- Iterating arrays with “**range**” (concise way)

```
a := [...]float64{1.1, 2, 3.0, 4}
for i, v := range a { //from 0 to the length
   fmt.Printf("position %d : value = %.2f\n", i, v)
}
```

- Multidimensional arrays

```
a := [3][2]string {
         {"a", "A"},
         {"b", "B"},
         {"c", "C"}, // comma is mandatory
     }
fmt.Println("a : ", a)
```



```
a :  [[a A] [b B] [c C]]
```

Golang – Slice

- « Slice » means « **slice of array** »
- Declaration : like an array but **without size**

```
[]type
```

```
// Array (fixed size)
a := [5]int { 0, 1, 2, 3, 4, }
```

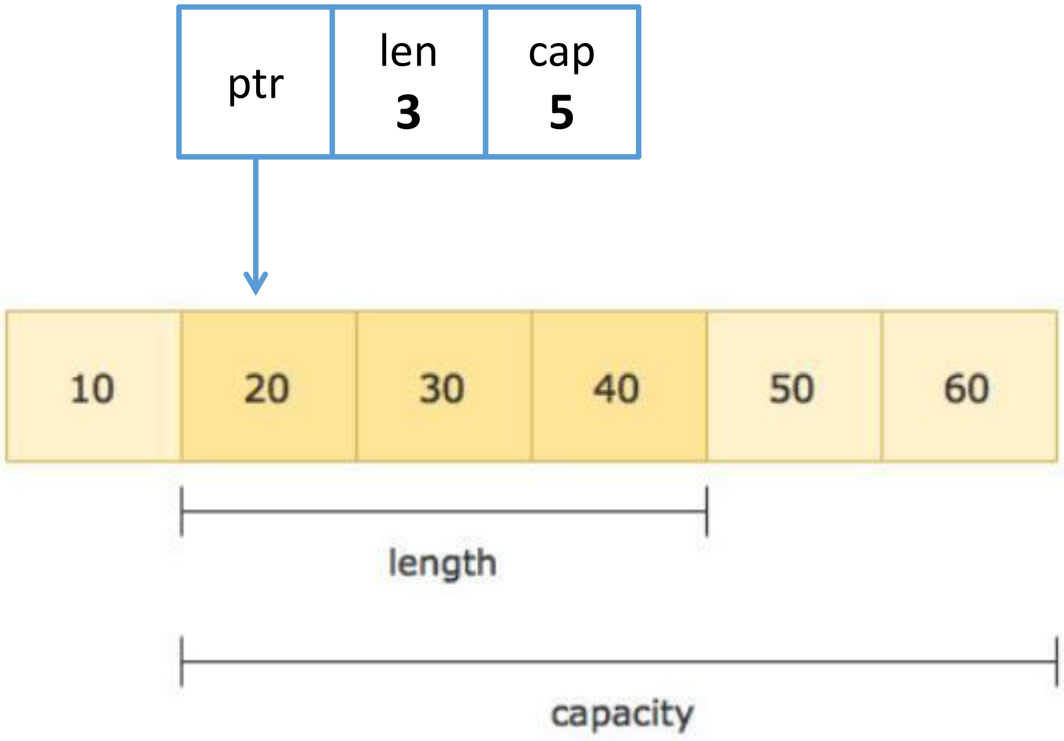
```
// Slice (no size)
var s []int  // nil, len = 0
```

Initialize a slice from an existing array

```
// Slice of array "a"

s = a // ERROR : array != slice

s = a[:] // Ok (all elements of ‘a’)

s = a[3:] // from index 3 to end

s = a[:4] // from index 0 to 4-1

s = a[1:3] // from index 1 to 3-1
```

Golang – Slice

- A slice is based on an **underlying array**
- A slice has both a **length** and a **capacity**.

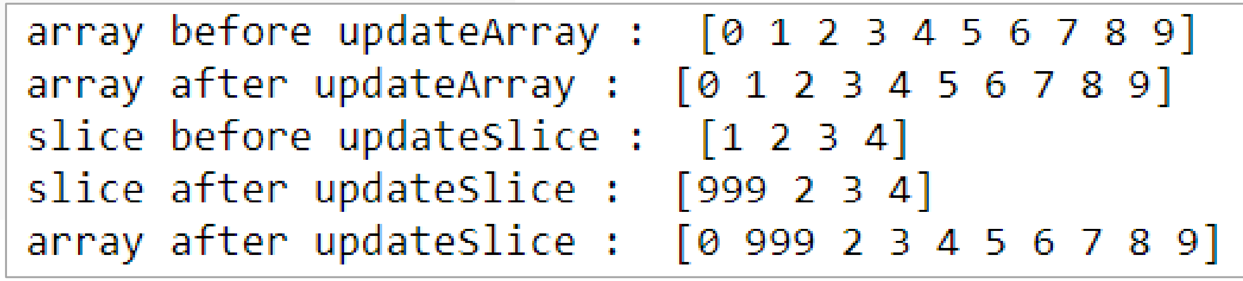


A slice does not own any data of its own. It is just a <u>representation</u> of the underlying array.

Any modifications done to the <u>slice</u> will be reflected in the underlying <u>array</u>.

Golang – Slice

```
func updateArray(array[10]int) {
    array[0] = 999  // No effect
}
func updateSlice(slice[]int) {
    slice[0] = 999
}
func main() {
    a := [10]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    fmt.Println("array before updateArray : ", a)
    updateArray(a) // passed by value
    fmt.Println("array after updateArray : ", a)

    s := a[1:5] // slice with underlying array
    fmt.Println("slice before updateSlice : ", s)
    updateSlice(s) // slice holds a pointer on the array
    fmt.Println("slice after updateSlice : ", s)

    fmt.Println("array after updateSlice : ", a)
    // array has been updated !
}
```

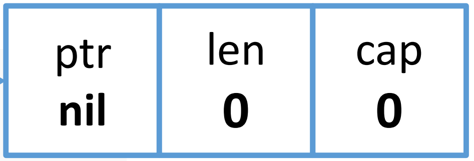


```
array before updateArray :  [0 1 2 3 4 5 6 7 8 9]
array after updateArray :  [0 1 2 3 4 5 6 7 8 9]
slice before updateSlice :  [1 2 3 4]
slice after updateSlice :  [999 2 3 4]
array after updateSlice :  [0 999 2 3 4 5 6 7 8 9]
```

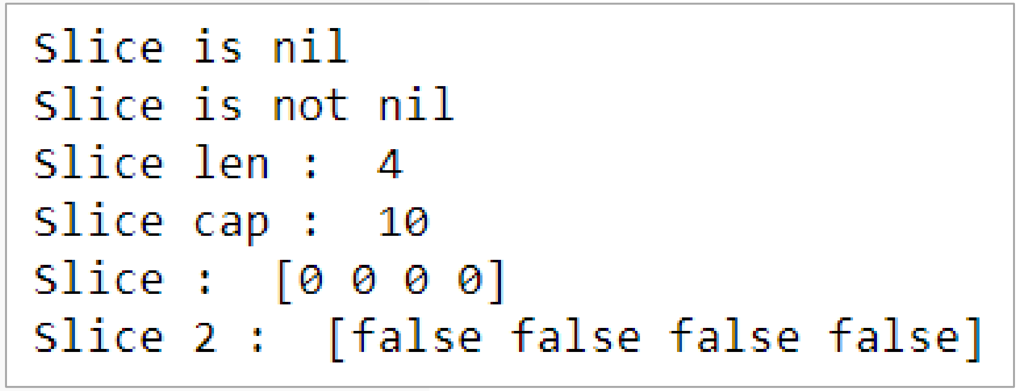
Golang – Slice : creation with “make”

```
a := make([]int, 5)     // len = 5, no cap => cap = len = 5

b := make([]int, 2, 5) // len = 2, cap = 5
```

```
var slice []int  // Declared not initialized
if slice == nil {
   fmt.Println("Slice is nil")
}
slice = make([]int, 4, 10)
if slice != nil {
   fmt.Println("Slice is not nil")
   fmt.Println("Slice len : " , len(slice) )
   fmt.Println("Slice cap : " , cap(slice) )
   fmt.Println("Slice : ", slice )
}

slice2 := make([]bool, 4, 10)
fmt.Println("Slice 2 : ", slice2 )
```



ptr	len	cap
nil	**0**	**0**



```
Slice is nil
Slice is not nil
Slice len :  4
Slice cap :  10
Slice :  [0 0 0 0]
Slice 2 :  [false false false false]
```

Golang – Slice : changes

Updating an element

```
s := []int{0, 1, 2, 3, 4}

s[2] = 222
```

Appending to a slice

```
s := []int{0, 1, 2, 3, 4} // len = cap = 5

s = append(s, 55, 66, 77) // len = 8 cap = 10
```

built-in function “**append**”

If the backing array of s is too small to fit all the given values a bigger array will be allocated

Delete element from a slice :

No function => do it yourself ! ☹

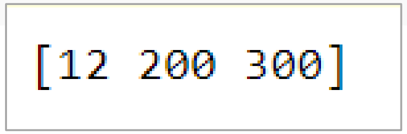
Golang – Array vs Slice

	Array	Slice
Length	Fixed	Variable
Capacity	Fixed (Capacity = Length)	Variable
Creation with ‘**make**’	No	Yes
Support ‘**append**’	No	Yes
Passed …	by value	by reference (for underlying array)

Golang – Function argument : Array ptr or Slice ?

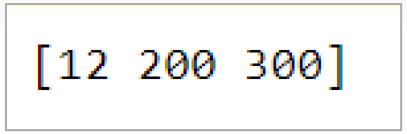
Function argument :
Do not pass a pointer to an array (it works but it’s not idiomatic)
=> use slice instead.

```
func modify(a*[3]int) {
    // (*a)[0] = 12
    a[0] = 12
    // a[0] is shorthand for (*a)[0]
}

func main() {
    a := [3]int{100, 200, 300}
    modify(&a)
    fmt.Println(a)
}
```



```
[12 200 300]
```

```
// works for any size of array
func modify(slice []int) {
    slice[0] = 12
    // change underlying array
}

func main() {
    a := [3]int{100, 200, 300}
    modify(a[:]) // slice
    fmt.Println(a)
}
```



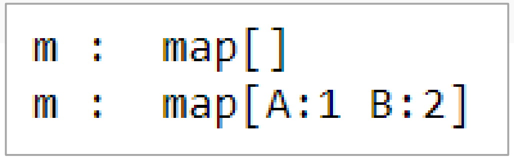
```
[12 200 300]
```

Golang – Map

- Map = built in type to manage "key-value" pairs

```
var m map [key_type] value_type
```

```
var m map[string]int // m = nil
fmt.Println("m : ", m)

m = make(map[string]int)

m["A"] = 1
m["B"] = 2
fmt.Println("m : ", m)
```



```
m :  map[]
m :  map[A:1 B:2]
```

```
m := make(map[string]int)
m["A"] = 1
m["B"] = 2
fmt.Println("m : ", m)
```

```
m := map[string]int {
 "A" : 1,
 "B" : 2,
}
fmt.Println("m : ", m)
```

```
fmt.Println("'B' value : ", m["B"] )
```

Golang – Map

```
v := m["X"] // v = ‘zero value’ if not found in map
```

- Check if exists

```
s := "C"
value, exists := m[s]
if exists == true {
   fmt.Println("Value of", s, "is", value)
} else {
   fmt.Println(s, "not found")
}
```

- Delete

```
fmt.Println("Length " , len(m) )
delete(m, "B")
fmt.Println("Length " , len(m) )
```

Golang – Map

- A map is passed by reference
  When a map is assigned to a new variable, they both point to the same internal data structure.
  Hence changes made in one will reflect in the other.
- A map can be “nil”
- 2 maps cannot be compared with “==“
  “==“ is usable only for “== nil”
- Iteration :

```
for key, value := range m {
  fmt.Printf(" %s : %d\n", key, value)
}
```

defer

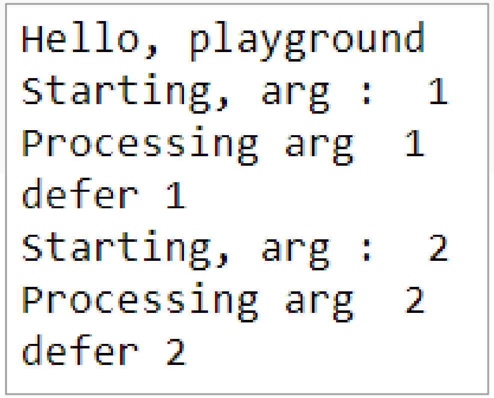
Error Handling

- Return error
- panic

Golang – Error handling - defer

A **defer** statement defers the execution of a function until the surrounding function returns.

```go
func foo(n int) {
   fmt.Println("Starting, arg : ", n)
   defer fmt.Println("defer", n)
   fmt.Println("Processing arg ", n)
}

func main() {
   fmt.Println("Hello, playground")
   foo(1)
   foo(2)
}
```



```
Hello, playground
Starting, arg :  1
Processing arg  1
defer 1
Starting, arg :  2
Processing arg  2
defer 2
```

```go
func foo(n int) {
   fmt.Println("Starting, arg : ", n)
   defer func() {
      fmt.Println("defer", n)
   }()
   fmt.Println("Processing arg ", n)
}

func main() {
   fmt.Println("Hello, playground")
   foo(1)
   foo(2)
}
```

Golang – Error handling - defer

The “defer” statement can be used **anywhere** in the function (even at the end) it will work if is evaluated before any “return”

A function can have **many “defer” statements**

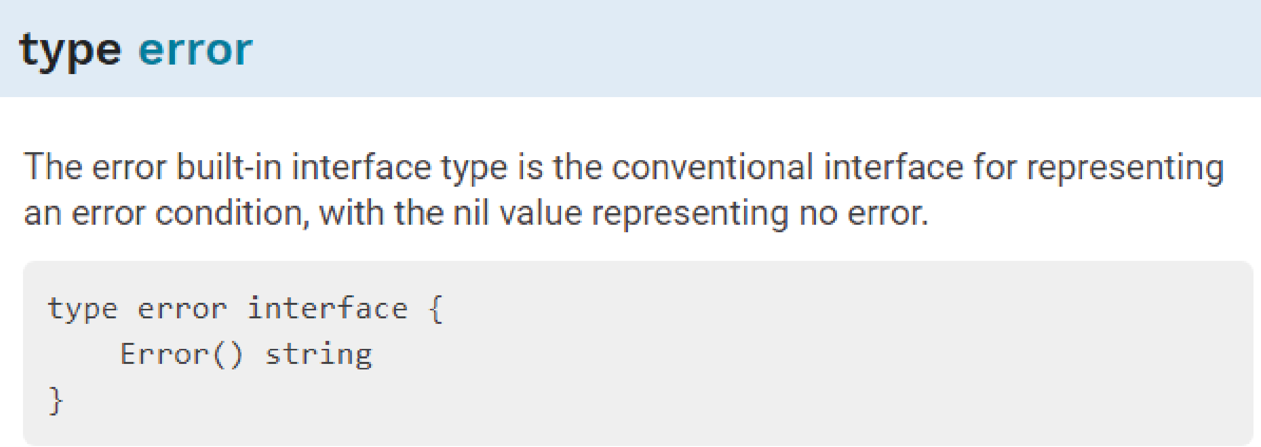
```
func foo(n int) {
  fmt.Println("Starting, arg : ", n)
  fmt.Println("Processing arg ", n)
  defer fmt.Println("defer 1 : ", n)
  defer fmt.Println("defer 2 : ", n)
  for i := 1 ; i <= 3 ; i++ {
    defer fmt.Println("defer #", i, " : ", n)
  }
  return
}
```

```
func main() {
  fmt.Println("---")
  foo(1)
  fmt.Println("---")
  foo(2)
  fmt.Println("---")
}
```

```
---
Starting, arg :  1
Processing arg  1
defer # 3  :  1
defer # 2  :  1
defer # 1  :  1
defer 2 :  1
defer 1 :  1
---
Starting, arg :  2
Processing arg  2
defer # 3  :  2
defer # 2  :  2
defer # 1  :  2
defer 2 :  2
defer 1 :  2
---
```

Golang – Error handling

- What's an error in Go ?
  any type implementing
  the “**error interface**”
  (**Error** function returning a **string**)



type error

The error built-in interface type is the conventional interface for representing an error condition, with the nil value representing no error.

```
type error interface {
        Error() string
}
```

- Functions **return errors**
  (functions and methods can return multiple values)
- Example :

```
func Open(name string) (file *File, err error)
```

```
f, err := os.Open("filename.ext")
if err != nil {
     log.Fatal(err)
}
// do something with the open *File f
```

Golang – Error handling - panic

- Panics are similar to C++ and Java exceptions, but are only intended for run-time errors
- A “**panic**” stops the normal execution of a goroutine

No object oriented but…

- Struct + Methods
- Interface
- Composition

Golang – Interfaces (declaration)

- An interface defines the “behaviour of an object”
  “Interface contract” = “what to do” (not “how”)
- Interface definition in Go :

```go
type MyInterfaceName interface {
    // List of functions to be implemented
    MyFunc1() return_type
    MyFunc2(arg)
    // etc
}
```

- Go has some predefined interfaces.
  Example : “Stringers” (package “fmt”)

```go
type Stringer interface {
    String() string
}
```

Golang – Interfaces (implementation)

```
type Stringer interface {
    String() string
}
```

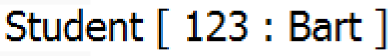
Implements

```
type Student struct {
      Id   int
      Name string
}

// Stringer interface  implementation
func (o Student) String() string {
      return fmt.Sprintf("Student [ %d : %s ]",
         o.Id, o.Name)
}
```

Implementation is not declared explicitly in source code !

Usage :

```
s := Student{123, "Bart"}
fmt.Println(s)
```



```
Student [ 123 : Bart ]
```

Golang – Interfaces (definition & implementation)

```go
// Shape interface definition
type Shape interface {
      Area() float32
}
```

```go
type Rectangle struct {
      width  float32
      height float32
}

// Shape interface implementation
func (r Rectangle) Area() float32 {
      return r.width * r.height
}
```

Implements

```go
r := Rectangle{10, 20}
fmt.Println("Rectangle Area : ", r.Area() )

var shape Shape = r
fmt.Println("Shape Area : ", shape.Area() )
```

Implementation is verified here

Golang – Interfaces (remarks)

- How to check interface implementation with the “structure” definition ?
- Just try to assign a “nil Rectangle” to a variable of type “Shape”

```
type Rectangle struct {
      width  float32
      height float32
}

// Check Rectangle implements Shape
var _ Shape = (*Rectangle)(nil)

// Shape interface implementation
func (r Rectangle) Area() float32 {
      return r.width * r.height
}
```

Implementation is verified here

Golang – Structure creation

- There’s no “constructor” in Go, but we can use functions to create structures

```go
type Rectangle struct {
        width  float32
        height float32
}

func NewRectangle(w, h float32) *Rectangle {
        return &Rectangle{width: w, height: h}
}
func NewRectangle2() *Rectangle {
        r := new(Rectangle)
        return r
}
func NewRectangle3(x float32) *Rectangle {
        r := new(Rectangle)
        r.width = x
        r.height = x
        return r
}
```

Reminder : a function name must be unique !

Different arguments
 => different names ☹

Other possibility :
the “builder” pattern

Golang – Structures and composition

- Go is not object-oriented and doesn’t support inheritance ☹
- So, think "Composition" instead of "Inheritance"

“Animal”
Generic class
that will be used in
other more
specialized classes

```
type Animal struct {
        Id   int
        Name string
}

func NewAnimal(id int, name string) *Animal {
        o := new(Animal)
        o.Id = id
        o.Name = name
        return o
}

func (o Animal) Eat() {
        fmt.Printf("Animal %s eats!\n", o.Name)
}
```

Golang – Structures and composition

```
type Animal struct {
    Id   int
    Name string
}
```

```
type Tiger struct {
        Animal
        Color string
}

func NewTiger(id int, name string, color string) *Tiger {
        o := new(Tiger)
        o.Id = id      // Animal
        o.Name = name // Animal
        o.Color = color
        return o
}

func (o Tiger) Run() {
        fmt.Printf("Tiger %s runs!\n", o.Name)
}
```

"Tiger"

A specialized class reusing "Animal"

Usage :

```
a := NewAnimal(12, "Felix")
fmt.Println(a)
a.Eat()
```

```
t := NewTiger(22, "Pluto", "White")
fmt.Println(t)
t.Eat()
t.Run()
```

Concurrency

- Go routines

```
// kind of lightweight threads
go myfunction(a,b)
```

- Channels

Not yet in this course…
See : https://medium.com/@trevor4e/learning-gos-concurrency-through-illustrations-8c4aff603b3

END