## for range loop

for i, v **:=** range s {

(loop body)

}

Normal for loop:

for i **:=** 0; i < len(s); i++ {

(loop body)

}

## String literals

It has 2 forms:

With quotes is **interpreted String literals**

x = “This is a string”

or **raw String literals** whit back quotes (**`**)

y = **`**This is a different

string: a “raw literals” in Golang.**`**

## Array initialization 1

In a newline Go required a last comma.

arr := [...]string{

"ready", "Get", "Go", "to"**,**

}

It isn’t necessary instead in one line initialization

arr := [4]string{"ready", "Get", "Go", "to"}

## Array initialization 2 ????

arr[i] = i + 1

It’s not allowed!

arr[i] = i + 1

This works.

## Map initialization

func main() {

patrons := make(map[int]string)

patrons[0] = "Terrence"

patrons[1] = "Evelyn"

fmt.Println(patrons)

}

With **make** Go require a declaration key indexing (mapName[mapKey]), equals and **no braces**.

func main() {

patrons := map[int]string{

0: "Terrence",

1: "Evelyn",

}

With this 2nd method the key word make is no more necessary but:

1. I need **braces**;
2. a **colon** instead of a equal sign;
3. a **comma** even in the last key-value as an end of line

var patrons = make(map[int]string)

is equivalent to

var patrons = map[int]string{}

This last line of code create an empty map using a map literal and the map is initialized.

## Anonymous struct

// declare the 'car' struct type

type car struct {

make string

model string

mileage int

}

// After create an instance of a car

newCar := car{

make: "Ford",

model: "taurus",

mileage: 200000,

}

Instead an anonymous struct: No type struct type!

newCar := struct {

make string

model string

mileage int

}{

make: "Ford",

model: "Taurus",

mileage: 200000,

}

## Interfaces

An **interface** is a collection of **method signatures** that a **Type** can implement (using **methods**). Hence **interface** defines (not declares) the behavior of the object (of the type Type).

The primary job of an interface is to provide only method signatures consisting of the **method name**, **input arguments** and **return types**. It is up to a Type (e.g. struct type) to declare methods and implement them.

## Variables declaration

*Keyword valueName Type*

Var myVar int

## Function

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | ***Signature*** | |  |
| *Function declaration* | *Function identifier* | *Parameter list*  *(optional)* | *Return type*  *(optional)* |  |
| **Function** | **myFunction** | **(i, j int)** | **int** | **{...}** |
|  |  |  | (require ***return***) |  |

## Wraparound

Wraparound means the number goes from its highest possible value to its smallest

possible value. Wraparound can be easy to miss when developing your code and can

cause significant problems to your users.