

**SAPIENZA, UNIVERSITY OF ROME**  
COURSE OF APPLIED COMPUTER SCIENCE AND ARTIFICIAL  
INTELLIGENCE (ACSAI)  
3RD YEAR, 1ST SEMESTER

---

# **WEB AND SOFTWARE ARCHITECTURE**



---

**NOTES BY LEONARDO BIASON**  
COURSE TAUGHT BY PROF. EMANUELE PANIZZI



**SAPIENZA**  
UNIVERSITÀ DI ROMA

**L**

## About these notes

Those notes were made during my three years of university at Sapienza, and **do not** replace any professor, they can be an help though when having to remember some particular details. If you are considering of using *only* these notes to study, then **don't do it**. Buy a book, borrow one from a library, whatever you prefer: these notes won't be enough.

## License

The decision of licensing this work was taken since these notes come from **university classes**, which are protected, in turn, by the **Italian Copyright Law** and the **University's Policy** (thus Sapienza Policy). By licensing these works I'm **not claiming as mine** the materials that are used, but rather the creative input and the work of assembling everything into one file.

All the materials used will be listed here below, as well as the names of the professors (and their contact emails) that held the courses.

The notes are freely readable and can be shared, but **can't be modified**. If you find an error, then feel free to contact me via the socials listed in my [website](#). If you want to share them, remember to **credit me** and remember to **not** obscure the **footer** of these notes.

## Bibliography & References

[1] TODO

The "*Web and Software Architecture*" course was taught in the Winter semester in 2024 by prof. Emanuele Panizzi ([panizzi@di.uniroma1.it](mailto:panizzi@di.uniroma1.it))

I hope that this introductory chapter was helpful. Please reach out to me if you ever feel like. You can find my contacts on my [website](#). Good luck!

Leonardo Biason

→ [leonardo@biason.org](mailto:leonardo@biason.org)

# CONTENTS

**CHAPTER 1** ▶ **APIs AND VERSION CONTROL** \_\_\_\_\_ **PAGE 1** \_\_\_\_\_

**CHAPTER 2** ▶ **BACKEND AND GO** \_\_\_\_\_ **PAGE 2** \_\_\_\_\_

# CHAPTER 1 APIs and Version Control

This is a test

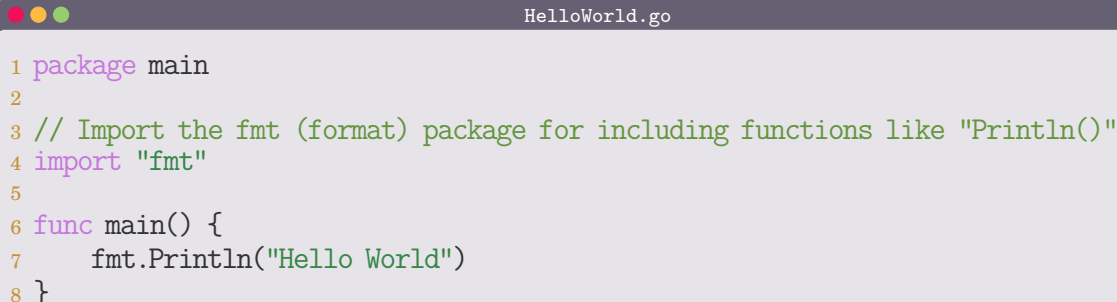
# CHAPTER 2

## Backend and Go

Go is a statically typed programming language created by Google, and it's made for building high-performance and scalable applications. Widely used in the Web, it allows to build powerful backends with high-level features such as **garbage collection** and **memory safety**. Its syntax is similar to the one of C, but it's way simpler. It's also more verbose though, so it's very similar to the choice that also Rust's creators took.

Go is structured in packages, where each package represents a collection of files and folders. A package is declared with the syntax `package <name>`. In a folder, only one package may be declared. Go requires a main package, which will contain the first functions to be executed.

Within a package, all the declarations are **global**: this means that a function declared in a file can be used in another file within the same package. In order to use other packages, we use the `import <package_name>` syntax. Let's see an example of Go code for a simple Hello World program:



```
1 package main
2
3 // Import the fmt (format) package for including functions like "Println()"
4 import "fmt"
5
6 func main() {
7     fmt.Println("Hello World")
8 }
```

We can now build our code (because Go produces a binary) with the following command:



```
go build HelloWorld.go -o HelloWorld
```

and then run it as a standard executable. By executing the binary, we obtain the following:



```
$ ./HelloWorld
```

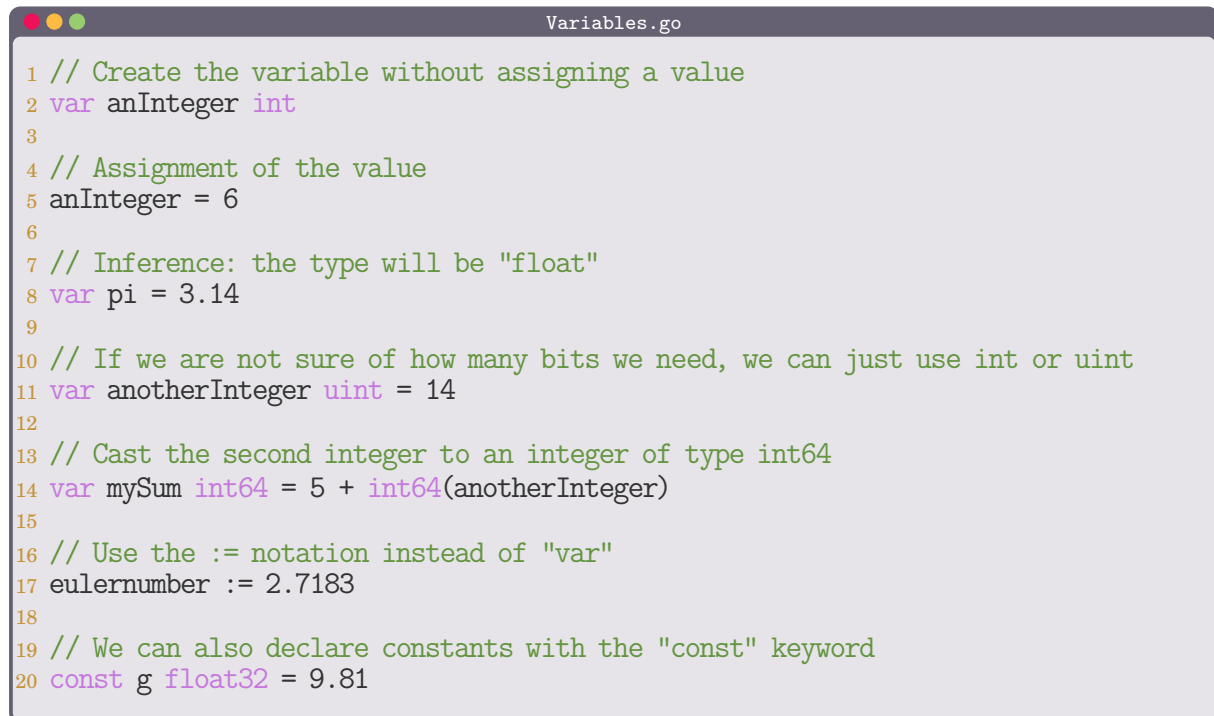
```
Hello World
```

Go is a statically typed language, so we are encouraged to write down the types of the

variables that we create for code security. Variables are defined either through the `var` keyword or through the `:=` operator.

Some types that are allowed in Go are `bool` for booleans, `intXX` and `uintXX` for respectively signed and unsigned integers (the `XX` stands for the number of bits that the number should take, so a value between 8, 16, 32 and 64), `string` for strings, `byte` for the bytes (it's an alias for `uint8`), `rune` for Unicode characters (has a size of 32 bits), `uintptr` for memory pointers, and so on and so forth...

Each variable is initialized to a specific value if no value is assigned. For most of the types, the standard value is 0, for strings, it's an empty string and for booleans it's `false`. Inference is also done if a type is missing. We can also declare constants if we want to. Here is an example of some variables:



```
1 // Create the variable without assigning a value
2 var anInteger int
3
4 // Assignment of the value
5 anInteger = 6
6
7 // Inference: the type will be "float"
8 var pi = 3.14
9
10 // If we are not sure of how many bits we need, we can just use int or uint
11 var anotherInteger uint = 14
12
13 // Cast the second integer to an integer of type int64
14 var mySum int64 = 5 + int64(anotherInteger)
15
16 // Use the := notation instead of "var"
17 eulernumber := 2.7183
18
19 // We can also declare constants with the "const" keyword
20 const g float32 = 9.81
```

We can also declare arrays which, similarly to C's arrays, all have a fixed length. Arrays are declared with the `var <name> [<size>]<type>` syntax. This syntax underlines how Go's arrays can store only one type of elements. So, for instance:

```

1 var myArray [10]int
2
3 // If we want to also assign some values, we have to do the following:
4 assignedArray := [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
5
6 // We can access to an array's value with the "[]" notation
7 var mySum = assignedArray[5] + assignedArray[3]
8

```

Go also uses the concept of slices, which allows to access to portions of the array just like in Python. The syntax is `[initial index (included):final index(excluded)]`, where the lowest initial index is 0 and the highest final index is the length of the array. Slices do not contain data, but just point to a section of the array.

There is a way to have dynamic arrays, and it's to use the slices mechanism. For instance, with `bools := []bool{true, false, false}` creates a dynamic array. But why does it use the slices? Because with the previous notation we are just creating an array of not known size and then we created a slice to that array.

Slices have also two important properties: **length** and **capacity**.

#### Length and Capacity of a slice

DEFINITION

The **length** of a slice denotes the number of elements in the slice.

The **capacity** of a slice denotes the number of elements that the sliced array contains.

Let's make a couple examples, to better grasp the concept: if we define an empty slice with `var a_slice []int`, then we have a slice with both length and capacity equal to 0. Now though, if we append one element with `a_slice = append(a_slice, 40)`, we now have that both the capacity of the underlying array and its length increased to 1. If we instead created a slice such as `var a_slice = []int{1, 2, 3, 4}`, then we would have a capacity and a length of both 4.

Go also provides a more granular way to control the creation of slices, and that's through the `make()` function. This function has the following syntax:

```

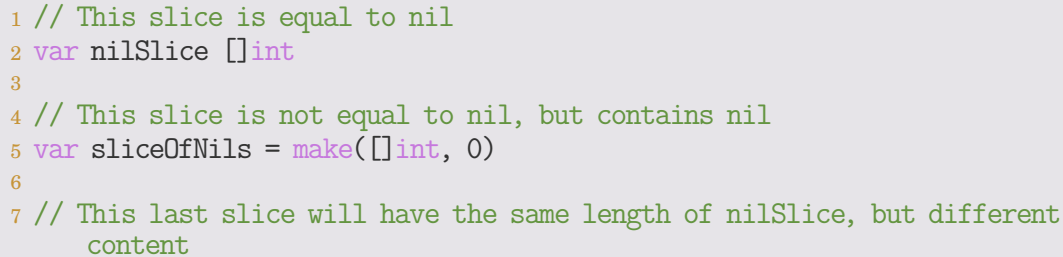
1 make(
2     []type,
3     length,
4     capacity
5 )

```

#### Parameters of `make()`:

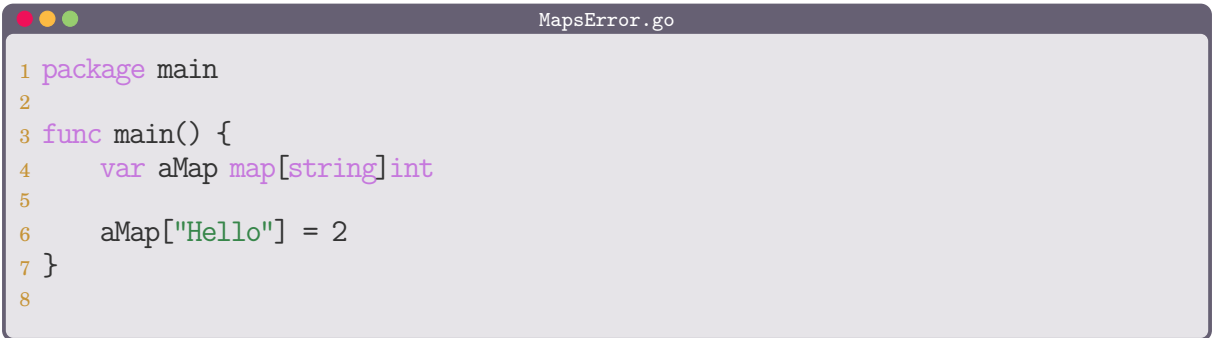
- `[]type`: the type of the slice;
- `length`: the length of the slice;
- `capacity`: the capacity of the slice.

For instance, with `var a_slice = make(int[], 0, 7)` we are creating a slice of capacity 7 and with a length of 0. Note that slices can assume a nil value (which is equivalent to C's NULL or Python's None) if they are declared but never assigned, but if they get declared through `make()` they will be equal to a slice of nils. A clearer example follows:



```
1 // This slice is equal to nil
2 var nilSlice []int
3
4 // This slice is not equal to nil, but contains nil
5 var sliceOfNils = make([]int, 0)
6
7 // This last slice will have the same length of nilSlice, but different
  content
```

We can also have the equivalent of Python's dictionary in Go, and they are called **maps**. A map can either be declared but not assigned (and this must be done with the `make()` function only) or declared and assigned. Why can we only declare with `make()`? Let's try to run the following code:



```
1 package main
2
3 func main() {
4     var aMap map[string]int
5
6     aMap["Hello"] = 2
7 }
8
```

If we try to execute this program, we will obtain the following error:



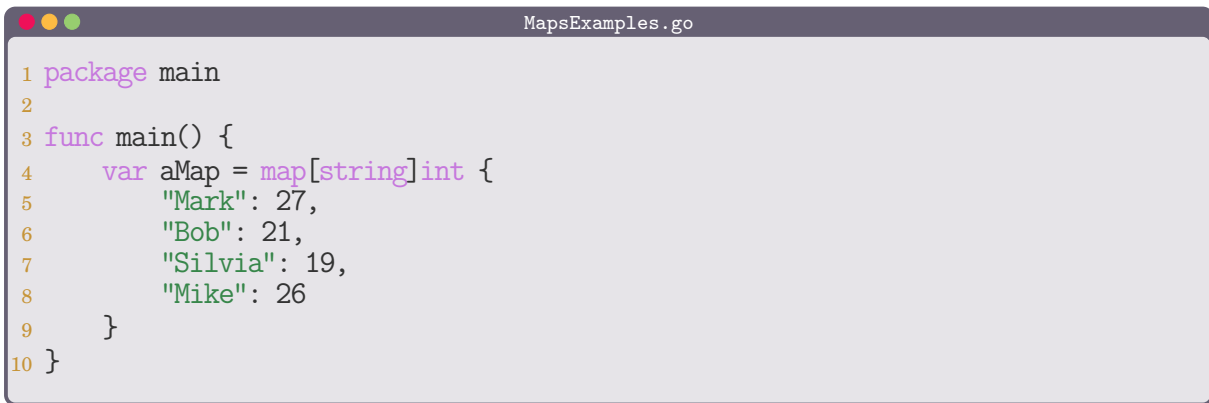
```
$ ./MapsError
```

```
panic: assignment to entry in nil map

goroutine 1 [running]:
main.main()
    /home/user/go/MapsError.go:8 +0x28
```

This is because the `var aMap map[string]int` instruction just declared the map, but never allocated it. So, in order to allocate it, we must use `make()`. We can then do the following:





```
1 package main
2
3 func main() {
4     var aMap = map[string]int {
5         "Mark": 27,
6         "Bob": 21,
7         "Silvia": 19,
8         "Mike": 26
9     }
10 }
```