Golang 2023

Alten Italia

## Github

<https://github.com/tomiok/course2023>

## Links with documentation

[Go tour](https://go.dev/tour/welcome/1)

[Go by examples](https://gobyexample.com/)

[Go packages](https://pkg.go.dev/)

[W3C](https://www.w3schools.com/go/)

[HTTP examples](https://go.dev/doc/tutorial/web-service-gin)

[Context](https://www.educative.io/answers/golang-context)

## Exercises

[Exercism](https://exercism.org/tracks/go)

[Gophercises](https://github.com/gophercises)

## Concurrency

[Channels](https://blog.logrocket.com/how-use-go-channels/)

[Concurrency patterns](https://go.dev/talks/2013/advconc.slide#1)

[Goroutines exercises](https://www.golangprograms.com/goroutines-and-channels-example.html)

[Velotio blog for channels](https://www.velotio.com/engineering-blog/understanding-golang-channels)

## Libraries used in the course

[Gorm](https://gorm.io/)

[Chi](https://go-chi.io/#/)

[Zerolog](https://github.com/rs/zerolog)

[Testify](https://github.com/stretchr/testify)

## Testing

[Unit testing](https://blog.jetbrains.com/go/2022/11/22/comprehensive-guide-to-testing-in-go/)

[Learn with tests](https://quii.gitbook.io/learn-go-with-tests/)

**What is GO?**

25 Reserved words

Statically typed, compiled language

Syntax similar to C++

Small memory footprint, it allows to deploy huge programs.

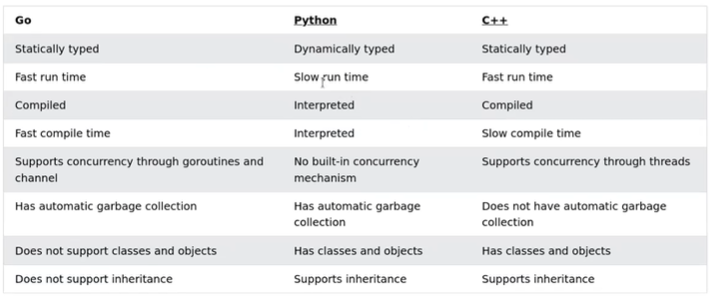
Not OOP, it’s procedural (Like C)

**What is used for?**

* Web-dev server-side (**MAIN REASON**: Backend)
  + Docker is developed in GOLang
  + Kubertenics is developed in GOLang
* Network-based programs
* Cross-platform enterprise applications
  + Compiler can compile for every OS
* Cloud-native dev

**Why?**

* Easy to learn
* Fast Runtime
* Concurrency
* Mem management (mainly done by the Garbage collector)
* Cross platform



First word in every GO file is a reserved word.

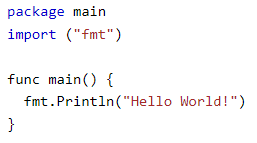
Or curly braces.

We have pseudo indentation.

**Syntax**

A Go file consists of the following parts:

* Package declaration
* Import packages.
* Functions
* Statements and expressions (always inside a function)
* Methods



***“fmt”***: Format



go mod init example.com/hello

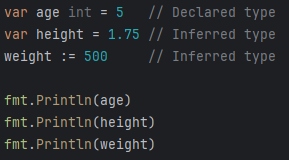
go run .\helloworld.go

go build .\helloworld.go

**Variables**

* int
* flaot32
* string
* bool

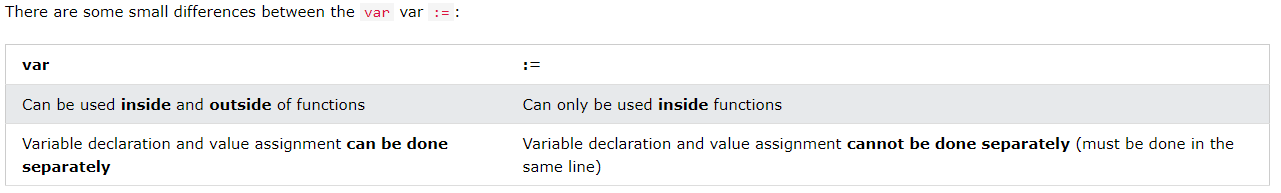
To declare a variable there are 2 ways:



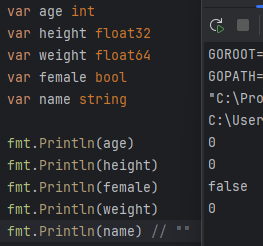
If the type is not declared, like in 2nd and 3rd example, it is **INFERRED**.

Everything I declare I must use it, otherwise it will have a compile error.

Only const and variables declared global, outside a function, don’t need to be used in order to compile.



**Default values**

****

**Constants**

Cannot change during runtime.



I can declare them in global scope, and I don’t need to use them in order to compile the file.

**Visibility**

There is no **public** or **private** but:

* **Exported**

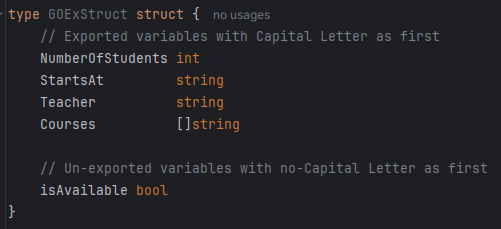
That filed or struct is visible outside the package (in the example main)

CAPITAL LETTER

* **Un-exported**.

The field or struct is not visible outside the package.

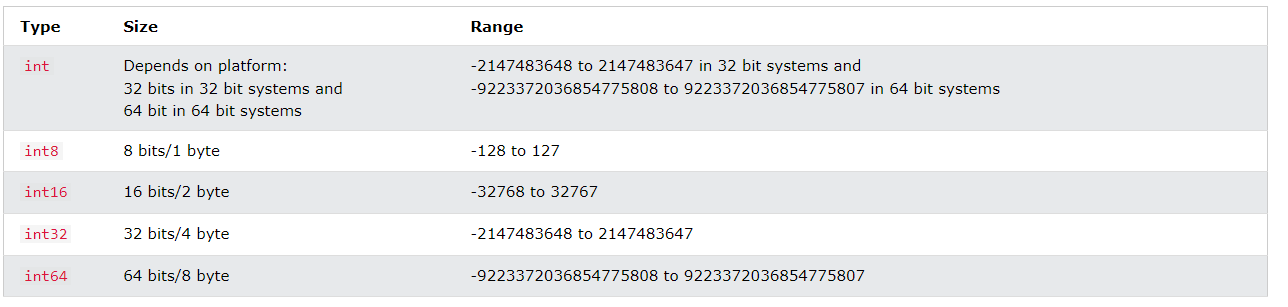
NOT CAPITAL LETTER (lower case)



**Datatypes**

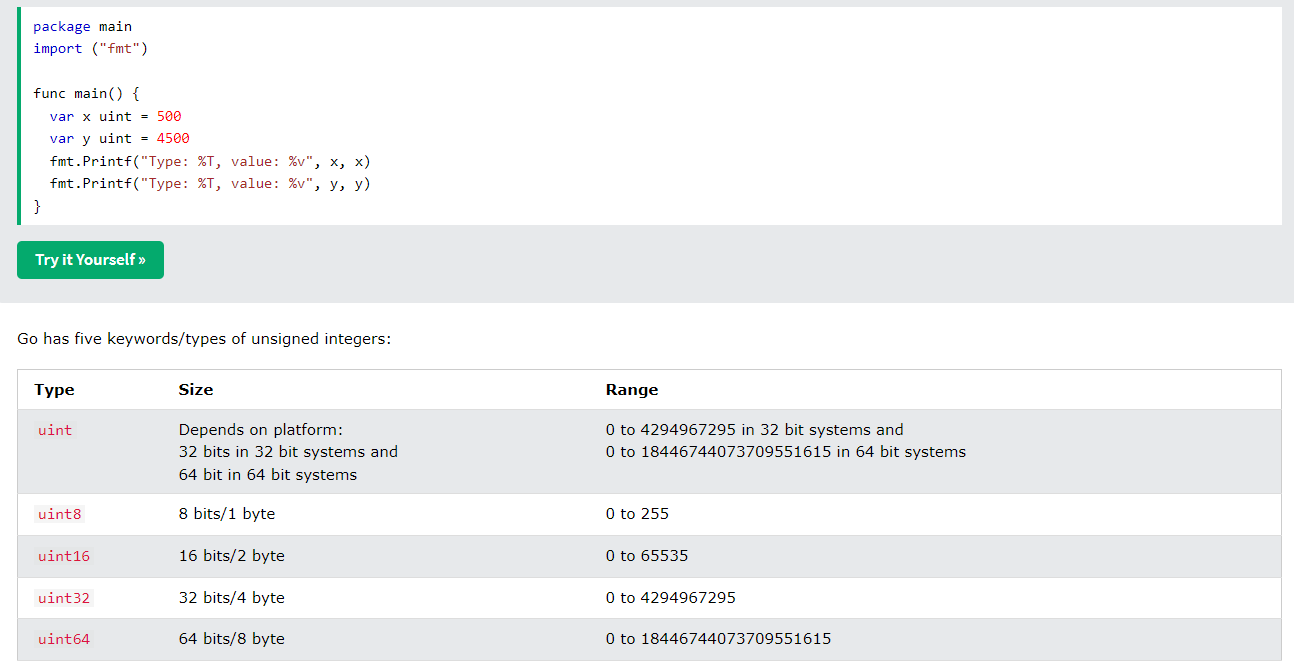
* **int**

Signed int



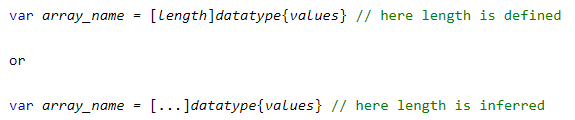
* **uint**

unsigned int



* **float32**
* **float64 (double)**
* **string**
* **bool**

**Arrays**

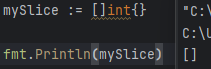


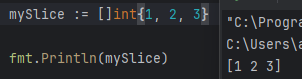
**Slices**

Similar to array, but more powerful and flexible.

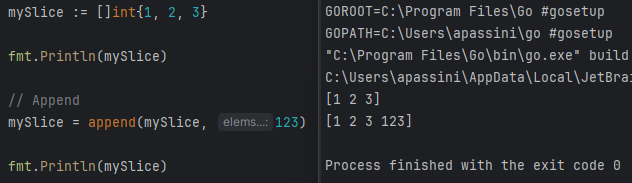
Flexible size, like a **C++** **vector** or ArrayList

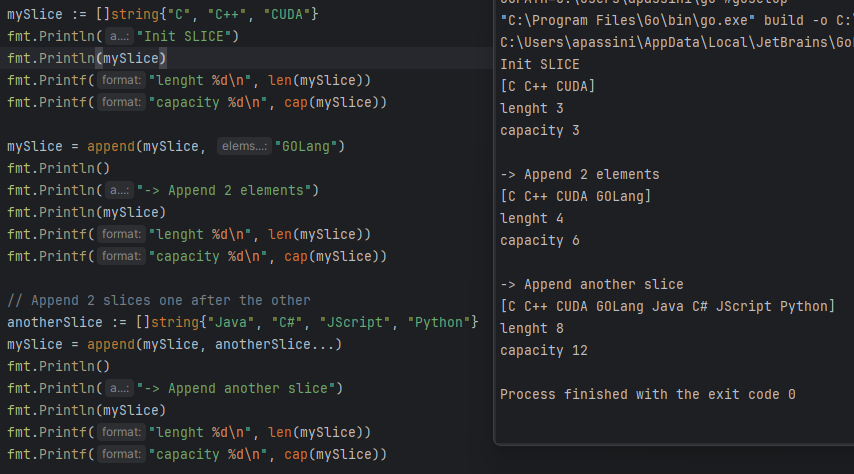






* **Append**

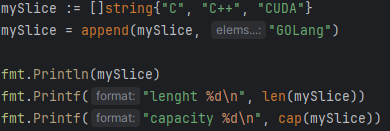




* **Length and Capacity**

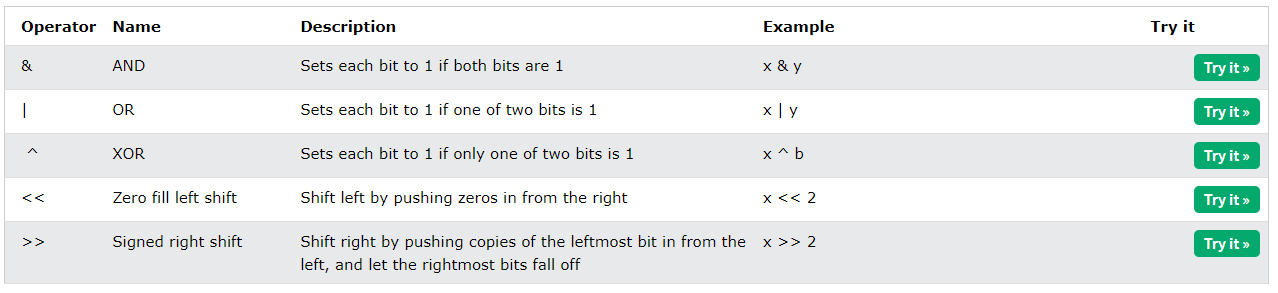
**Length:** is the actual length of the slice, the number of elements

**Capacity:** the number of elements the slice can grow or shrink to



**Operators**

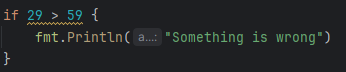
* **Bitwise**

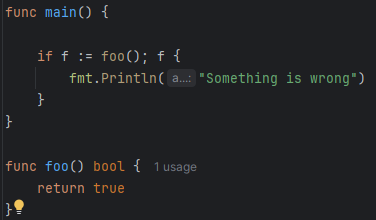
****

**Conditions**

* 1 for TRUE
* 0 for FALSE

No ternal condition



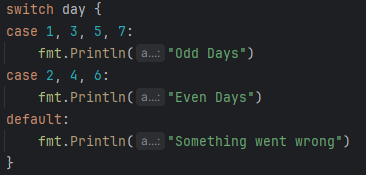
****

**Switch**

* Single case:

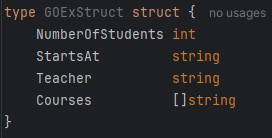


* Multi case:

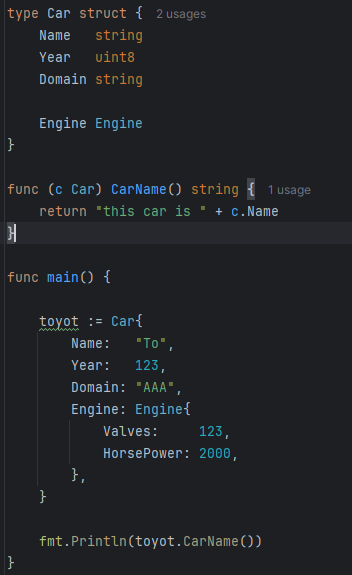


**Struct**

GOLang does not have Classes but structs (like C).

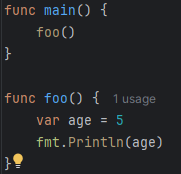


* Methods

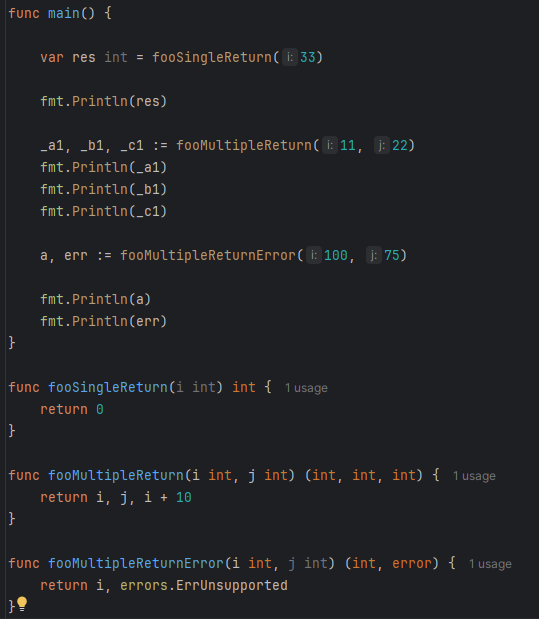


**Functions**

Example:

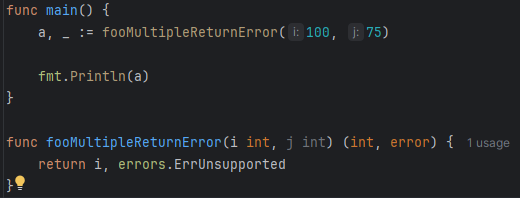
****

**Multi return value**



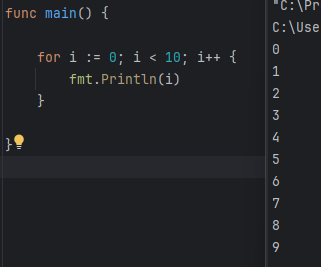
**Ignore values**

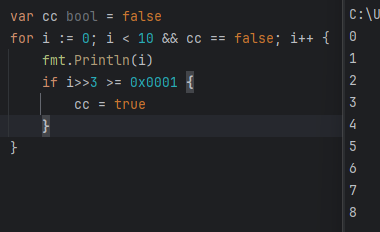
Using “\_” we can ignore the value



**Loops**

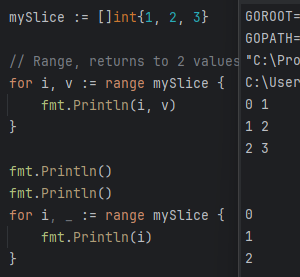
Only “**for**” loops, no “**while**” or “**do\_while**”





* **Range**

Returns index and the value a that index.



* **Len**

Length, the number of elements

* **Cap**

Capacity, the size that the slice can shrink or expand to

**Pointers**

* **Syntax**

Engine \*Engine

varName \*varType

Reference

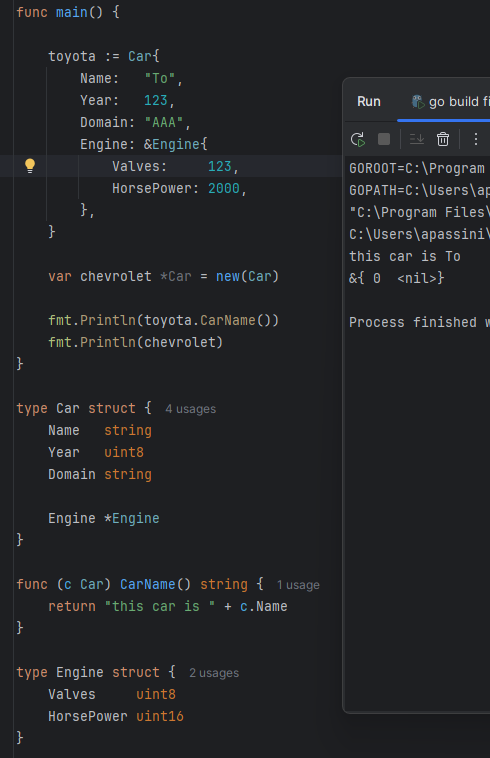
&varName

**Allocating with NEW**

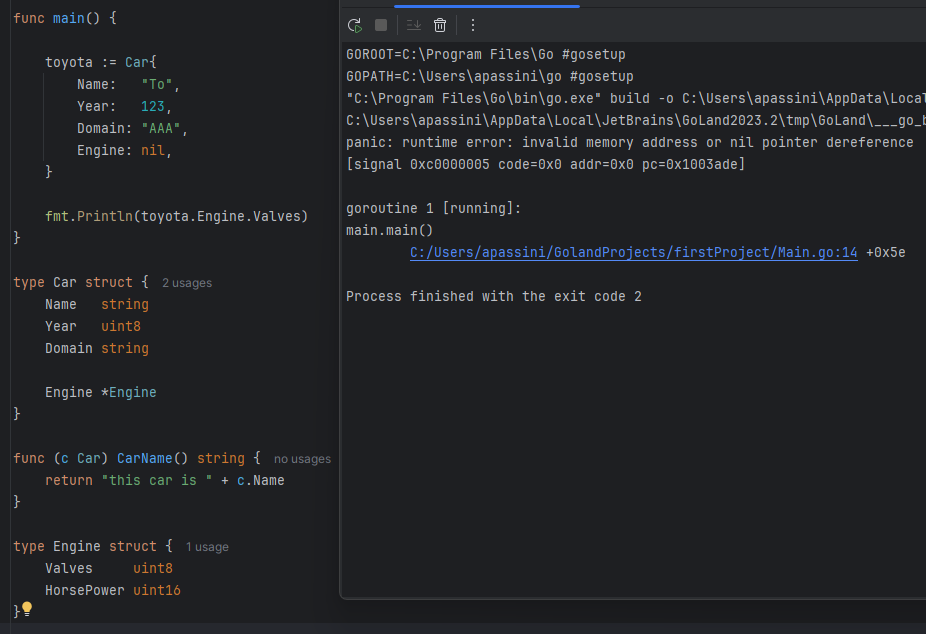
Allocating the struct and returning a pointer.

Allocate the memory at zero:

* Pointer: nil
* int: 0
* string: “”



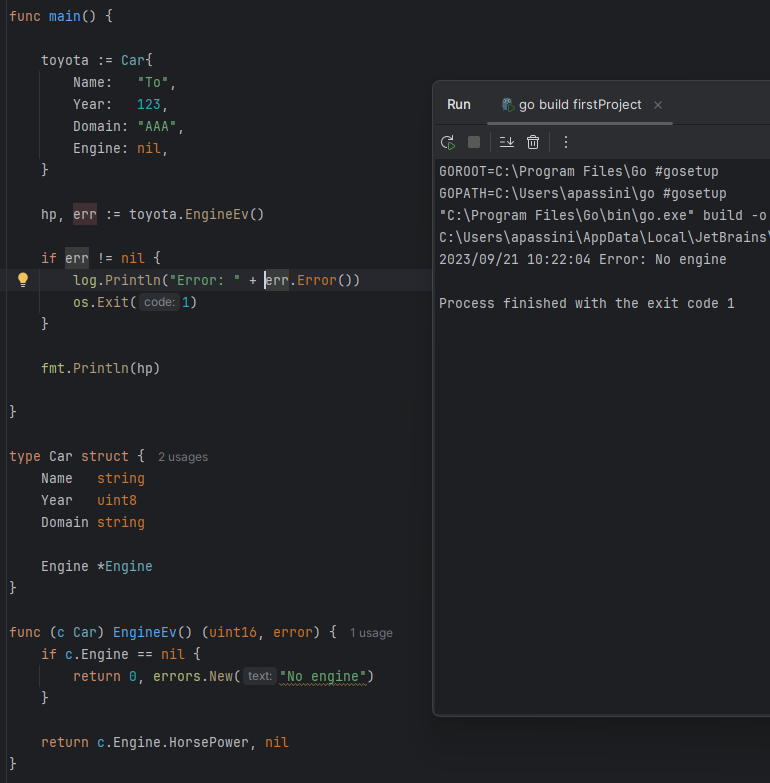
If you try to access a nil pointer, an error is thrown:



To avoid this problem, check the value with nil.

If we are implementing a method, we usually make it returns the value that we want and an Error. When we call the method, check if error is != nil.

**Error** is a native type.



**Memory Allocation (** [**https://go.dev/doc/effective\_go#allocation\_new**](https://go.dev/doc/effective_go#allocation_new) **)**

Not necessarily is a heap allocation.

In GO stack is dynamically sized by the compiler, what cannot be traced into the scope a single function will be allocated on the HEAP, where the garbage collector is.

* **New**

Allocate a struct or a user type.

Return a pointer.

Do not initialize the memory but it zeros it.

***new****(T) allocates zeroed storage for a new item of type T and returns its address, a value of type \*T*

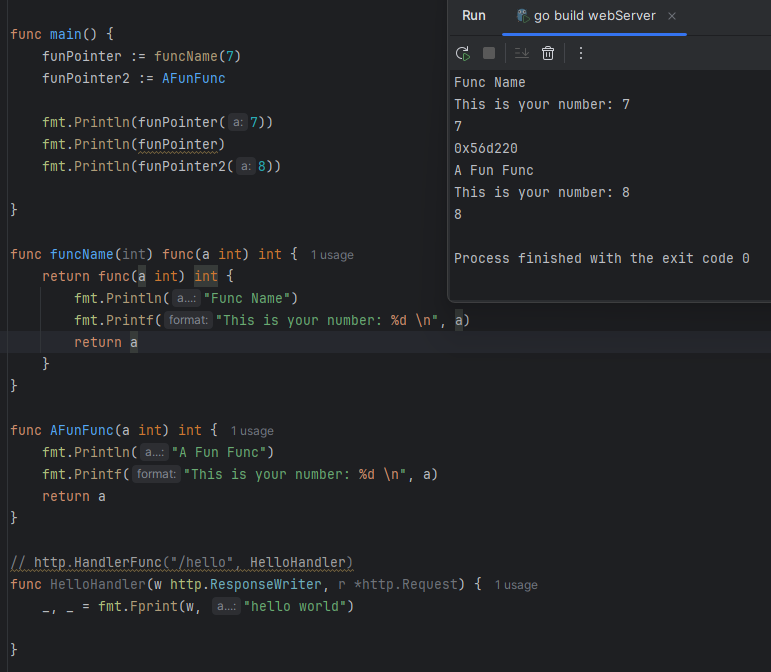
* **Make**

***make****(T, args) serves a purpose different from new(T). It creates slices, maps, and channels only, and it returns an initialized (not zeroed) value of type T (not \*T)*

**Function Pointer**

There are 2 ways to use a function pointer:

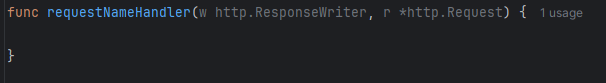
1. assign to a variable the function without parentheses.
2. Let the function return a function with specific syntax.



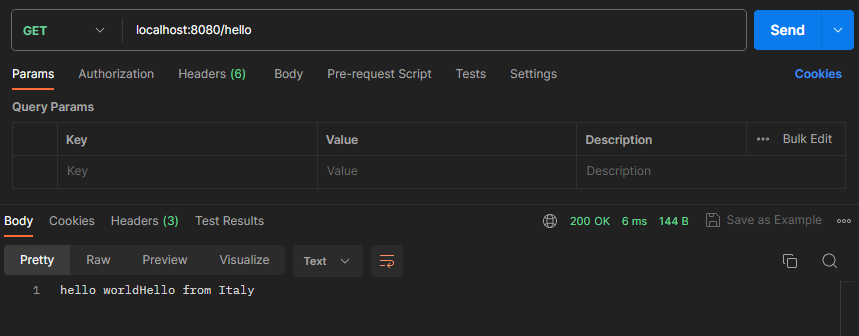
**Web Server in GO**

* **Handler**

A function that manages an HTTP request

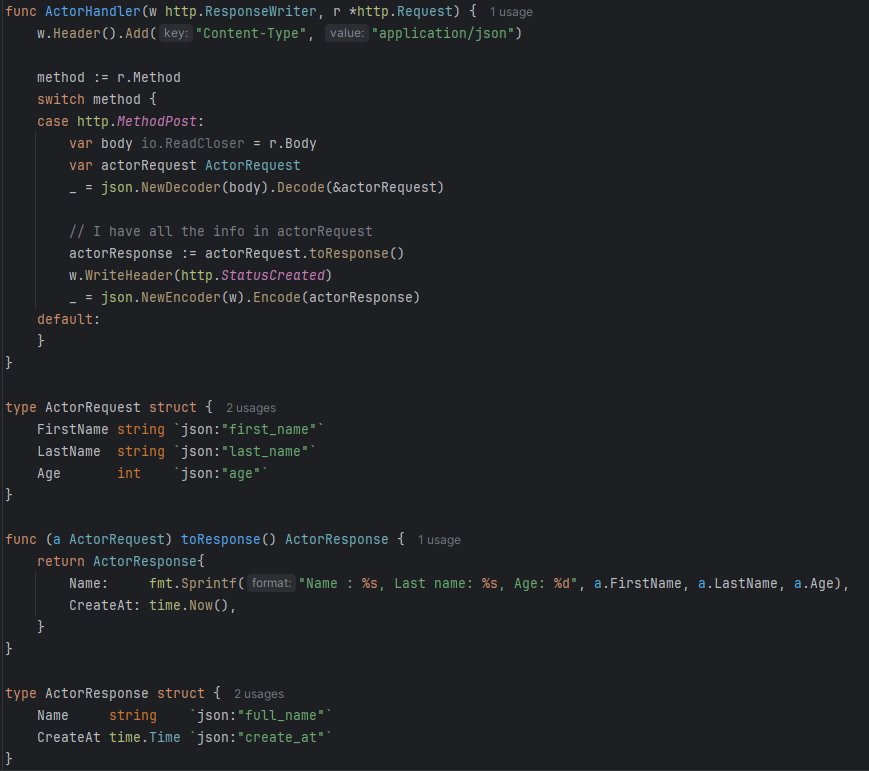








**JSON**

****