Introduction to GO

(for Java developers)

Who am I

I'm Matias Laino, working in Globant since 2012, joined as a Microsoft .net developer, but now a Java developer for some years.

I'm from Uruguay, currently working for Glovo, in Barcelona, Spain, since 2019.

Why we're here

I had already developed an internal tool in python™ to ease my development tasks when I needed a new one, I needed to write a new one which required writing to Kinesis hosted in LocalStack, and doing serialization of documents in a binary format (Avro) and I had another frustrating experience with Python due to its weak typing and duck typing, I decided to give Go a chance as a replacement for Python, and ended up enjoying it a lot, I thought others might find it as cool and useful as I did, so I did this presentation.

I do not claim to be a Go expert :) and I will be comparing Go's approach on things to Java's in some parts to make it easier to understand a few concepts, at least it helps me!

This is a quick introduction to Go, in no way an in-depth course.

What's Go?

Go, sometimes called Golang, is a **statically typed**, **strongly typed** and **compiled** language developed by Google and released in 2009.

It's **multiplatform**, favors **readability** (well, it's the intention at least), and has a **very high performance** for **networking** and **multiprocessing**, surpassing python and nodejs in many benchmarks.

Currently there's even a compiler that converts Go to Javascript, to run in browsers.

Most importantly, its mascot is a nameless gopher.

Hello world!

```go
package main
import "fmt"

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

There is always a package called main, and it always contains the main function.

Functions are first-class citizens in Go, they do not need to be wrapped in a class.

```
$ go run hello_world.go

Hello world!
```

go run will not “just” execute the file, it will actually compile it, store it in a temp folder, run it, and then delete the binary when it’s done.

If we used go build hello_world.go instead, it would have generated an executable file for the target system.

Installing

To install a go application in our local system, we use the go install command.

```
$ go install hello_world.go
```

This will compile and install the application into the $HOME/go folder.
The runtime is compiled as part of our binary.

Third party Go tools can installed directly from their repositories, no need for Maven Central or similar.

```
$ go install github.com/rakyll/hey@latest
```

Some standard Go tools

Go is extremely opinionated in how the code is written, no more arguments about indentation (how many spaces, or tabs) or braces, code that does not follow the strict syntax will not compile, some other idioms are not enforced but strongly encouraged, as they are convention.

- **go fmt** - Formats your code
- **goimports** - Validates your imports, sorts them, and tries to guess when there is a missing import.
- **golint** - Linting
- **go vet** - Vetting, detects some errors in your code such as wrong number of parameters passed to a formatting function.

Some of these tasks are run automatically by the IDEs and plugins upon file save.

Some IDEs

Visual Studio Code (free)

GoLand (JetBrains, commercial)

Variables

Go has several ways of declaring variables, let’s see for example how to declare a Boolean variable.

All of the following are equivalent:

```go
var myBooleanVariable bool = false
```

Explicitly declares a new variable of type boolean, and initializes it with false.

```go
var myBooleanVariable bool
```

Explicitly declares a new variable of type boolean, not initial value is specified, so it will use the Zero value for the type, this is basically the default value.

```go
var myBooleanVariable = false
```

Explicitly declares a new variable of type boolean, uses type inference to determine the variable type based on the initial value.

```go
myBooleanVariable := false
```

Shorthand syntax, declares a new variable and uses type inference based on the initial value.
This is the most common way to declare vars!

We redefine the value of a variable the usual way:

```
var x = "old value"
x = "new value"
```

We can do fun stuff such as **declaring and initializing multiple variables** in a single line:

```
var a, b bool
var x, y, aBooleanVariable, aStringVariable = 1, 2, true, "Hey there!"
```

Declare them in a **declaration list**:

```
var (
    v1 bool
    v2, v3 = true, "another string var"
)
```

Or **declare** new variables **AND redefine** the values of existing variables:

```
aStringVariable, aNewStringVariable := "New value for old var", "New value for NEW var"
```

Note that to use **:=** for multiple variables, **there must always be a new variable on the left side**.
This may have unintended consequences with Shadowing, it may create a new variable when you think you're reusing an old one.

Zero value

When a variable is declared but not initialized, it will default to the Zero value of the type, this is very different from what C, Java, C#, JS, etc do.

Built-In Primitive Types

Boolean - the Zero value is false.

```go
var myVar bool = false
var myVar bool
```

Integer - the Zero value is 0, same as all other numeric types.
There's a bunch of integer types: int32, int64, uint32, uint64, byte, etc; depending on whether it's signed or unsigned, and the size.
The type "int" (and "uint") depends on the target platform (will probably be int64 since that's what your CPU most likely is).
The type "byte" is an alias for "uint8"

```go
var myVar int = 0
var myVar int
```

Floating Point - the Zero value is also 0.
We have float32 and float64 to choose.
Interestingly, a non-zero float divided by 0 will yield +Inf or -Inf

```go
var myVar float64 = 0
var myVar float64
```

Strings - the Zero value is an empty string.
All Strings in Go are UTF-8. There's also the "rune" type which represents a single char, it's an alias for uint32

```go
var myVar string = ""
var myVar string
```

Complex numbers - Go supports these, it's unlikely that you'll use them, but hey, it's free!

Operators

Most, if not all operators will be very familiar.

+	sum	integers, floats, complex values, strings
-	difference	integers, floats, complex values
*	product	integers, floats, complex values
/	quotient	integers, floats, complex values
%	remainder	integers

==	equal
!=	not equal
<	less
<=	less or equal
>	greater
>=	greater or equal

&	bitwise AND	integers
|	bitwise OR	integers
^	bitwise XOR	integers
&^	bit clear (AND NOT)	integers

&&	conditional AND	p && q is "if p then q else false"
||	conditional OR	p || q is "if p then true else q"
!	NOT	!p is "not p"

<<	left shift	integer << unsigned integer
>>	right shift	integer >> unsigned integer

There are a couple of new, weird ones, though, we'll see them after!

Literals

This is just like in other languages, this is when we write a number, character, or string.

Integer literals: 4, 1_000_000 (decimal with underscore separators, for legibility),
0xAB10 (hexadecimal),
0b001001,
012 (octal).

Floating point literals: 12.5,
6.03e23.

Rune literals: ‘a’ (unicode char),
‘\141’ (8 bit octal number),
‘\x61’ (8 bit hex number)),
‘\u0061’ (16 bit hex number)),
‘\U00000061’ (32 bit unicode hex number),
‘\n’, ‘\t’, ‘\”, ‘\”’, ‘\\’

Strings: “Just a string”, “A string with\na new line or \”quotes\””

Unused variables, and throwaway variable names

An important note to make about variables.
A Go program that contains an **unused variable** (declared and/or initialized, but not used after), **will not compile**.
It does the same thing for imports!

Casting

Go doesn't support automatic casting like some other languages, all **casting needs to be explicit**.
You cannot, for example, automatically cast an integer to a float, the cast needs to be explicit.

```
var x int = 5
var y float64 = 5.5

var a = float64(x) + y
var b = x + int(y)
```

All casts are of the form **type(variable)**, they will work as long as the types are compatible.

```
var castedVariable = targetTypeName(valueInAnotherType)
```

Constants

Constants in Go are just a way to name literals, they are not like the final keyword in Java, we cannot declare an immutable variable with them. **A constant's value needs to be known at compile time**.

A constant can be **typed or untyped**.

```go
const x int = 10
```

Since in go there is no automatic casting, having the constant be typed means that it can only be used in expressions of the same type as the constant.

An untyped constant is more flexible and allows us to use it with variables of different type.

```go
const x = 10
```

So we can do the following without casting:

```go
var y int = x
var z float64 = x
var d byte = x
```

Arrays

Arrays are rarely used directly in our go programs.
An array is declared like this:

```go
var x [3]int
```

We can construct initializing all values explicitly:

```go
var x = [3]int{10, 20, 30}
```

We can also initialize only specific array positions:

```go
var x = [10]int{1, 4: 20, 6}
```

Unspecified indexes will be **initialized with the Zero value** of the type

And we can skip the size, it will be inferred by the compiler:

```go
var x = [...]int{1, 2, 3}
```

We compare arrays very easily:

```go
var x = [...]int{2,3,4}
var y = [3]int{2,3,4}

fmt.Println(x == y)
```

Assignment is as we'd expect:

```go
x[2] = 5
```

To get the size of an array, we use the built-in len function

```go
len(x)
```

Go does something very different than other languages when it comes to arrays: **the size is part of the type**.

It means that **[3]int is a different type than [4]int**, a function cannot just return "an array", it's always going to be "an array of size N". It's **not possible to convert arrays of different sizes**.

For dynamic size collection, Go uses something called slices.

Slices

Slices are **similar to ArrayLists in Java or .NET**.
They're basically a **collection that expands as we add items to it**, under the hood it will use an array as a backing store.

```go
var x = []int{10, 20, 30}
```

Note that there is no size in [], and no three dots.

```go
var x = []int{1,2,3, 9: 22}
```

Creates a slice with values: 1, 2, 3, 0, 0, 0, 0, 0, 0, 22

We assign values to indexes the same way we do for arrays:

```go
x[3] = 10
```

Slices **cannot be compared with ==**, we can only compare them with their Zero value, which is **nil**.
nil is not like null, nil means "no value", but it's also a value of the same type of the slice.

```go
var x = []int;
fmt.Println(x == nil) // prints true
fmt.Println(len(x) == 0) // prints true
```

Accessing a slice out of the constructed range causes a runtime panic (kinda like an unhandled exception, it’s an unrecoverable error).

```go
var slice =  []int{1,2,3}
slice[3] = 4 // Panic!
```

To grow a slice, we need to use the append function.

```go
slice = append(slice, 4, 5, 6) // Adds the values 4,5 and 6 to the slice, making it larger.
```

To append the values from one slice or array to another, we use the … (ellipsis) operator.

```go
var slice2 = []int{4, 5, 6}
slice = append(slice,
slice2...)
```

Go is Pass by Value

Go parameters, like Java's, are always passed by value, this means that the value that the function receives when it runs is a copy of the original. This means that the slice passed to the append function in the previous example is actually a copy of the original. The param slice is not modified by the append function, so we always need to assign the result back to the variable.

There is a way to pass the "real" variable value, we'll see how later!

cap is a built-in function like len, but **returns the capacity of a slice** rather than the length.

The **length of the slice is the amount of items stored in the slice**.
The **capacity is how many items it can store before having to resize itself**. If you push items past the capacity of the slice, the slice will need to create a new backing array to store more information and copy the contents of the old backing array to the new one (this is analogous to how Java or .NET do it with their ArrayLists (that's why it's called ArrayList, as opposed to LinkedList, for example)).

When we declare a slice, by default the capacity is set to the amount of items we initialize it with.

```go
var slice1 []string
```

cap(slice1) → 0	len(slice1) → 0

```go
slice2 := []string{"string1", "string2" }
```

cap(slice2) → 2	len(slice2) → 2

What if we append?

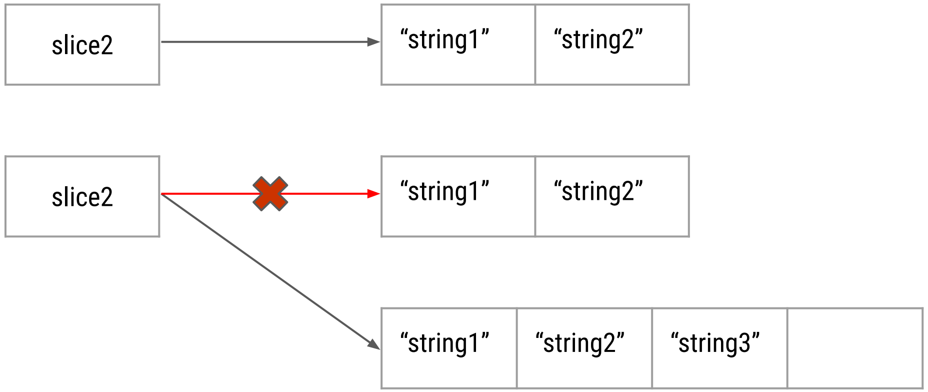
```go
slice2 = append(slice2, "another string")
```

cap(slice2) → 4	len(slice2) → 3

The capacity **was increased automatically by the runtime to accommodate another item**. The length was increased as well.
Capacity of the slices grow by expanding the backing array to twice it's original size, or half of it, depending on the memory usage.

```go
slice2 := []string{"string1", "string2" }
```

```go
slice2 = append(slice2, "another string")
```



The append will cause the runtime to create a new backing array, copy the contents over, then have the slice point to it instead. The old array will be cleared by the garbage collector at some point.

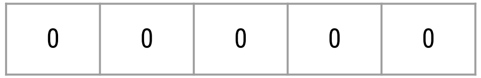
Yes! **Go has a garbage collector as part of its runtime**.

Make

We can tell the Go runtime to **create a slice with a given length**.

```go
slice3 = make([]int, 5)
```

cap(slice3) → 5	len(slice3) → 5

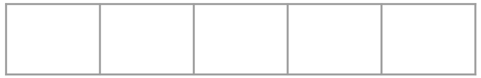


Since the length is how many items there are in an array/slice, initializing a slice with length N will not create a empty slice, but instead will create one initialized with N zero-value elements, in the case of an int slice, the zero value is 0.

If we don't want to initialize the elements, but just specify the capacity, we can also do it with make.

```go
slice4 = make([]int, 0, 5)
```

cap(slice4) → 5	len(slice4) → 0

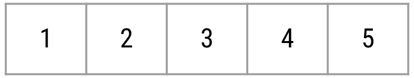


Never pass a length lower than the capacity, your program may fail in runtime (the compiler probably will pick it up, but it might not depending on your code).

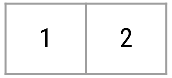
Slicing a slice

We can create a slice *from* a slice.

```
a := []int{1, 2, 3, 4, 5}
```



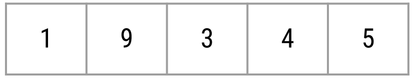
```
b := x[:2] //equivalent to x[0:2]
```

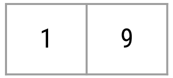


But here’s where slices start to get funny, let's overwrite the second item in the first slice.

```
a[1] = 9
```

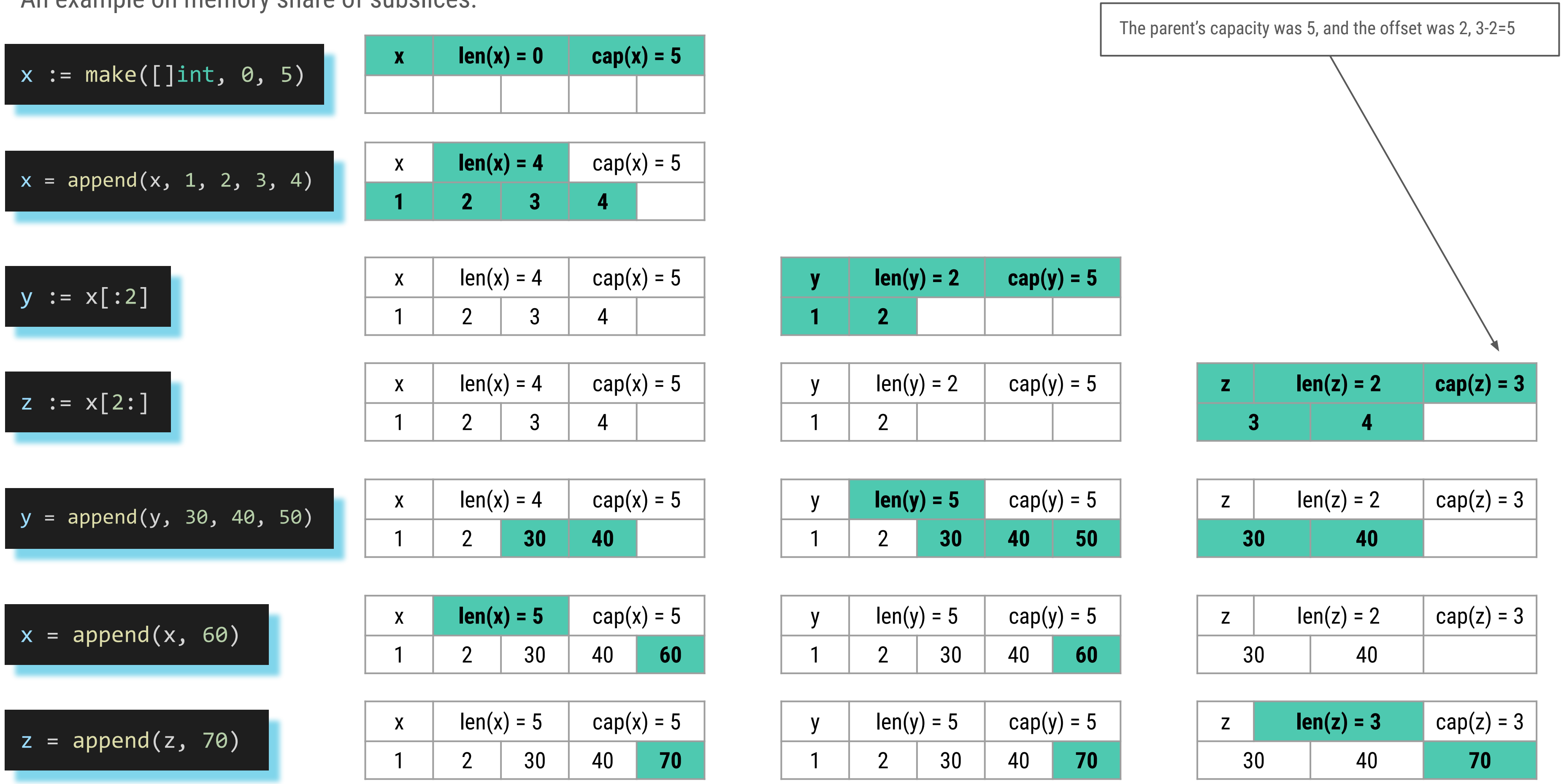
The slices look like this after the assignment:





Slices of slices share memory! Be very careful when modifying slices after they’ve been sliced.
A sub-slice’s capacity is equals to the parent’s capacity minus the starting offset.

An example on memory share of subslices.



```go
x := make([]int, 0, 5)
```

x	**len(x) = 0**		**cap(x) = 5**	

```go
x = append(x, 1, 2, 3, 4)
```

x	**len(x) = 4**		cap(x) = 5	
1	**2**	**3**	**4**	

```go
y := x[:2]
```

x	len(x) = 4		cap(x) = 5	
1	2	3	4	

y	**len(y) = 2**		**cap(y) = 5**	
1	**2**			

```go
z := x[2:]
```

x	len(x) = 4		cap(x) = 5	
1	2	3	4	

y	len(y) = 2		cap(y) = 5	
1	2			

z	**len(z) = 2**	**cap(z) = 3**
3	**4**	

```go
y = append(y, 30, 40, 50)
```

x	len(x) = 4		cap(x) = 5	
1	2	**30**	**40**	

y	**len(y) = 5**		cap(y) = 5	
1	2	**30**	**40**	**50**

z	len(z) = 2	cap(z) = 3
30	**40**	

```go
x = append(x, 60)
```

x	**len(x) = 5**		cap(x) = 5	
1	2	30	40	**60**

y	len(y) = 5		cap(y) = 5	
1	2	30	40	**60**

z	len(z) = 2	cap(z) = 3
30	40	

```go
z = append(z, 70)
```

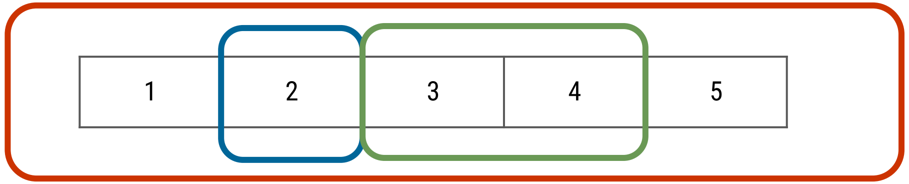
x	len(x) = 5		cap(x) = 5	
1	2	30	40	**70**

y	len(y) = 5		cap(y) = 5	
1	2	30	40	**70**

z	**len(z) = 3**	cap(z) = 3
30	40	**70**

To better understand subslices, it might be helpful to visualize them as different windows viewing different parts of the same data.

```
x := []int{1, 2, 3, 4, 5}
y := x[1:2]
z := x[2:4]
```



Full slice expressions

Full slice expressions help prevent some unwanted overwrite behavior like we saw in the prior example.

A full slice expression includes the last index of the original slice's capacity which is available to the subslice, it's a mean to control how much memory is shared.

```
y := x[:2:2]
z := x[2:4:4]
```

Changing the subslice will still change the original slice, but if you append a new value, it will create an array for itself.
I don't think this makes using subslices particularly simple, but it's there if you want it.

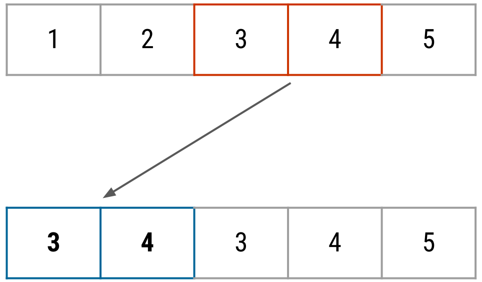
Copying slices

We can **create new slices and then copy the values from the old slice** into the new slice, without sharing memory.
The built-in **copy** function copies from one slice to another.

```
copy(destinationSlice, originSlice)
```

The origin and destination can even be the same slice:

```
mySlice := []int{1, 2, 3, 4, 5}
copy(mySlice[0:], mySlice[2:4])
fmt.Println(mySlice)
```



Maps

Maps are similar to other languages.
The Zero value of a map is nil.

```go
var nilMap map[string]int
```

Reading from a nil map always returns the zero value of the value type (in the case of int, it’s 0). Writing to a nil map causes a panic (unrecoverable runtime error).

To declare a useable map, we need to use a map literal:

```go
personNamesById := map[int]string{}
personNamesById[333] = "Someone"
personNamesById[555] = "Someone Else"

fmt.Println(personNamesById[666]) //prints an empty string
fmt.Println(personNamesById[333]) //prints “Someone”
```

We can declare a non empty map literal, and grow it from there.

```go
personNamesById := map[int]string{
    333: "Someone",
    555: "Someone else",
}
```

This last comma is not a mistake, all comma separated lists always end in a comma

Maps use arrays as storage, similarly to slices, therefore **they also have length and capacity**.
We can create a map of a given length and capacity with make.

```go
personNamesById := make(map[int]string, 5)
fmt.Println(len(personNamesById)) // will print the amount of key-value pairs in the map.
```

Maps are not comparable, we cannot use ==.

We delete from a map by means of the delete function.

```go
delete(myMap, key)
```

Comma ok idiom

Reading from a map using a non existent key will return the zero value of the map, if it’s a map of int values, it will always return a 0.

How can we tell whether a key exists in the map?
This is another place where Go is both weird, and cool.

Functions in Go can return more than one result, and they often do.
Lots of functions return both the expected result AND a boolean variable that indicates success (or failure). This is called the “**comma ok idiom**”.

```go
personNamesById := map[int]string{
    333: "Someone",
    555: "Someone else",
}

person, ok := personNamesById[777]

if !ok {
    fmt.Println("The person does not exist")
}
```

Structs

Go doesn't have classes, it has structs. They're really similar with one key difference: there is no class hierarchy, no inheritance. And wait until we get to interfaces, oh man!

```go
type person struct {
    name string
    age int
}

var somePerson person

anInitializedPerson := person{
    "Matias",
    35,
}

personWithOnlyAge := person{
    age: 22
}
```

No, these are not comma separated, for some reason!

Remember that declaring a variable without initializing it will result in a variable with value equal to the Zero-value of the variable type. In this case **the Zero-value of a struct type is a struct with all its fields set to their Zero-values**. Here it'd be a person struct with an empty string for name, and 0 as age.

There are no constructors, however a struct literal expression is kinda one. As you may guess, you pass all of its fields in the order they're declared and that initializes the struct.

There is no difference between
`var p person`
and
`var p person = person{}`

We access structs fields as you'd expect: `person.age`, `person.name`

Anonymous Structs

There exists the possibility of declaring an anonymous struct, this is used for (un)marshalling JSON and some unit tests.

```go
var book = struct{
    name string
    author string
} {
    name: "Dune",
    author: "Frank Herbert",
}
```

Equality and Type Conversion

We can't compare two structs of different types with ==, and there is no .equals method to override in Go.
However you can assign one struct variable to another of a different type IF both structs have the same fields, with the same types, and declared in the same order.

We can, then, use **type conversion** if we want to assign from one struct to another, or if we want to compare them. This applies to anonymous structs too.

```go
type notAPerson struct {
    name string
    age  int
}
var p1 = person{
        "I'm a person",
        44,
    }
var p2 = notAPerson {
    "I'm a person",
    44,
}

fmt.Println(p1 == person(p2)) // prints true
```

Blocks

In Go, **a block is a place where declarations occur**.
Functions, constants, variables and types declared **outside of a function** exist in the **package block**.
Import statements exist in the **file block**.
Within functions, **each set of {} defines a new block**.

Identifiers defined in a block exist only within it and within any child blocks, this is not dissimilar to other languages.

When we have a declaration with the same name as an identifier in a containing block, we say that we've **shadowed** the identifier.

```go
func main() {
    x := 10
    if x > 5 {
        fmt.Println(x) // Prints 10
        x := 5         // Declares a new variable with the same name as the original x, shadowing it! := only reuses 
variables defined in the current block
        fmt.Println(x) // Prints 5
    }
    fmt.Println(x) // Prints 10, the original variable was never modified
}
```

There is a linter to help detecting unintentional shadowing: $ go install golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow@latest

Conditionals

```
if condition {
    ...
} else if anotherCondition {
    ...
} else {
    …
}
```

A neat feature is that variables may be declared that will be visible only for the scope of the conditional.

```
var people = map[int]string{
    1: "Gianina",
    2: "Gonzalo",
    3: "Pablo",
    4: "Canary",
    5: "Matias",
}

if personName, ok := people[4]; ok {
    fmt.Println("Person found! " + personName)
}
```

Loops

In Go there is only for.
The **traditional** for:

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

The “**while**” for (wtf didn’t they just call it while…):

```
a := 100
for a > 0 {
    a--
}
```

The “**while true**” for:

```
for {
    // run indefinitely
}
```

And finally the “**foreach**” for:

```
values := [...]int{1,2,3,4,5,6,7}
for i, v := range values {
    fmt.Println("Index of the current iteration: ", i)
    fmt.Println("Value of the current iteration: ", v)
}
```

Sometimes we are not interested in the index variable, but remember **Go does not allow unused variables**, so we’re forced to declare a variable in a way that tells the compiler “I’m not gonna use it”.

```
for _, v := range values {
    fmt.Println("Value of the current iteration: ", v)
}
```

_ is the identifier for a throwaway variable, it's called the **Blank Identifier**.

We can **iterate over a map’s collection of key-value pairs**:

```
for k, v := range myMap {
    ...
}
```

Map key values’ order is not defined so never assume it. In fact, the order of the keys will always change whenever queried, this is a security feature.

However, to help debug. fmt.Println always prints a map’s key values in ascending sorted order.

Important note: the iteration variables in foreach-loops always contain copies of the values in a collection.

```go
people := []person{
    {"Jack", 10},
    {"Bob", 44},
}

for _, v := range people {
    v.age *= 2
}

fmt.Println(people) // Prints [{Jack 10} {Bob 44}], their age did not change.
```

Finally, Go supports the **break and continue keywords, they behave the same as in other languages**, however there is also support for loop labels, which are kinda similar to GOTO labels, but they're rare, so we won't see them now.

If you don't know or don't remember: break stops the execution of the current loop and executes the next line after the loop definition; continue stops the execution of the current loop and runs the next iteration.

Switch

```go
words := []string{"a", "bunch", "of", "words",
    "supercalifragilisticexpialidocious"}
for _, word := range words {
    switch size := len(word); size {
    case 1, 2, 3, 4:
        fmt.Println(word, "is a short word!")
    case 5:
        wordLen := len(word)
        fmt.Println(word, "is exactly the right length:", wordLen)
    case 6, 7, 8, 9:
    default:
        fmt.Println(word, "is a long word!")
    }
}
```

Can declare variables in the switch declaration, similar to conditionals and loops. Notice here that the switch is over the size variable, which is an int, it can be done over any other comparable type.

Multiple values in a single case

Braces are not necessary, they're implied. The same goes for the break statement, it's not necessary, there is no fallthrough by default unless you use a specific keyword.

Cases with empty blocks of code will just do nothing, it's equivalent to just having a break in other languages

Blank Switch

```go
number := 4

switch {
case number == 0:
    fmt.Println("Number is 0")
case number == 1:
    fmt.Println("Number is 1")
case number%2 == 0:
    fmt.Println("Number is even")
default:
    fmt.Println("Number is uneven")
}
```

Not performed over a single variable

Arbitrary boolean condition

You can put any conditions you want in a blank switch statement.

Functions

```go
func sqr(number int) int {
    return number * number
}
```

Go doesn't support function overloading :(

But it does support variadic input parameters (the ellipsis):

```go
func addAll(numbers ...int) int {
    var result int
    for _, num := range numbers {
        result += num
    }

    return result
}
```

Go supports multiple return values, like we saw before:

```go
func divAndRemainder(numerator int, denominator int) (int, int, error) {
    if denominator == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
    return numerator / denominator, numerator % denominator, nil
}
```

Go has **named return values**:

```go
func divAndRemainder(numerator int, denominator int) (result int, remainder int, err error) {
    if denominator == 0 {
        result, remainder, err = 0, 0, errors.New("cannot divide by zero")
        return
    }
    result, remainder, err = numerator/denominator, numerator%denominator, nil
    return
}
```

Go functions are **first class citizens**, and are thus treated as any other kind of value, you **can pass them around** and reference them the same way you do any other declared name.

```go
func add(i int, j int) int { return i + j }
func sub(i int, j int) int { return i - j }
func mul(i int, j int) int { return i * j }
func div(i int, j int) int { return i / j }
```

```go
var opMap = map[string]func(int, int) int {
    "+": add,
    "-": sub,
    "*": mul,
    "/": div,
}
```

We can declare function types to simplify the map declaration and reuse them in different parts of the code, this is similar to delegates in C#.

```
type arithmeticFunction func(int,int) int

var opMap = map[string]arithmeticFunction {
    // ...
}
```

These are used for documentation, code reuse, and later for interfaces.

Closures

Similar to lambda functions Java or C#, we can declare anonymous functions and pass them around to other functions, or call them immediately if we want to.

```
// sort by age
sort.Slice(people, func(i int, j int) bool {
    return people[i].Age < people[j].Age
})
```

We can also have our functions return other functions (these are called functors), assign them to variables, anything you can do with a value.

Defer

Defer is similar to the finally keyword in Java/C#, and it's used a lot for cleaning up resources.

```go
file, err := os.Open("./hello_world.go")
if err != nil {
    log.Fatal(err)
}
defer file.Close()

buffer := make([]byte, 2048)
for {
    count, err := file.Read(buffer)
    if err != nil {
        if err != io.EOF {
            log.Fatal(err)
        }
        break
    }
    fmt.Println(string(buffer[:count]))
}
```

We can defer functions or method calls. These will run when the containing function ends.

Defer runs after the return statement, and we can defer multiple functions, they will be executed in LIFO order.

```go
func funcWithMultipleDefers() {
    defer func() {
        fmt.Println("Deferred first")
    }()
    defer func() {
        fmt.Println("Deferred second")
    }()
    defer func() {
        fmt.Println("Deferred third")
    }()

    fmt.Println("Main function finished!")
}

func main() {
    funcWithMultipleDefers()
}
```

Output is:

```
Main function finished!
Deferred third
Deferred second
Deferred first
```

It might be useful for the deferred function (that we mentioned are usually cleanup functions) to know the return value of the containing function. To do this we need to use named return values.

```go
func multiplyByFive(param int) (result int) {
    defer func() {
        fmt.Println("The containing function returned ", result)
    }()

    result = param * 5
    return result
}
```

Pointers

In Go **all function parameters are passed by value**, which means that they are **always a copy of the original** variable, if you change the parameter's value inside the function, the original variable will remain unchanged. This is true also for arrays, and only kinda true for slices; so **if we pass a huge array as an argument, the entire array will be copied** before entering the function, this is **very inefficient**.
To very quickly sum up what a pointer is: **a pointer is a variable that contains a memory address**, accessing that memory address will take us to the actual value that the pointer points at.

Go is garbage collected, meaning that memory is managed by the garbage collector and not us, so that makes pointers less painful than in C, we don't need to free memory retained by a pointer..

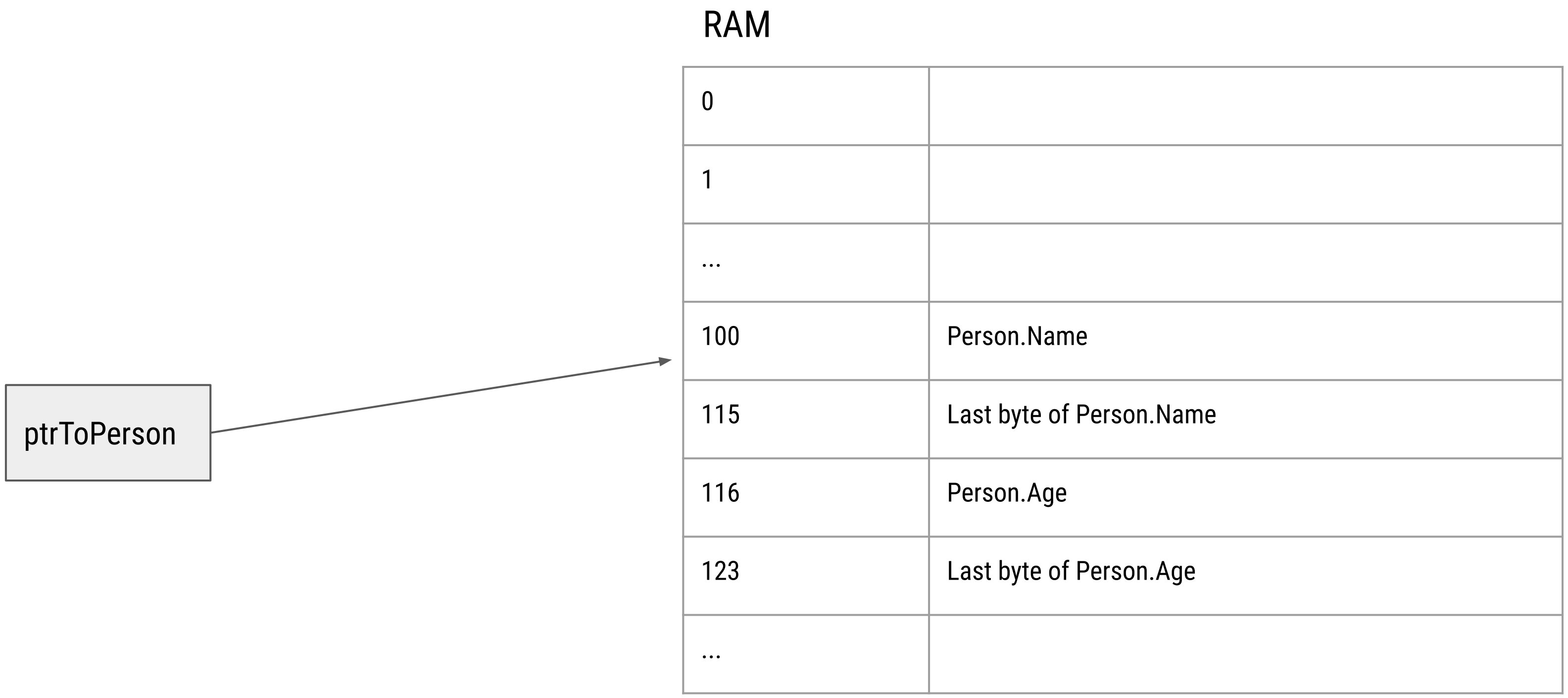
Pointer syntax in Go, however, is mostly the same as in C.

```go
var ptrToPerson = &Person {
    "PersonName"
} //Declares a new pointer variable of type person, declares a struct literal and assigns the memory address to the pointer.

var personName = *ptrToPerson.Name // The asterisk is the dereference operator, it means “go to the memory address contained by the variable ptrToPerson”.
```

The Zero-value of a pointer is nil.

Unlike C, the syntax doesn't change when accessing a pointed object's fields, it's always a dot.



0	
1	
...	
100	Person.Name
115	Last byte of Person.Name
116	Person.Age
123	Last byte of Person.Age
...	

Pointers are used for performance reasons only

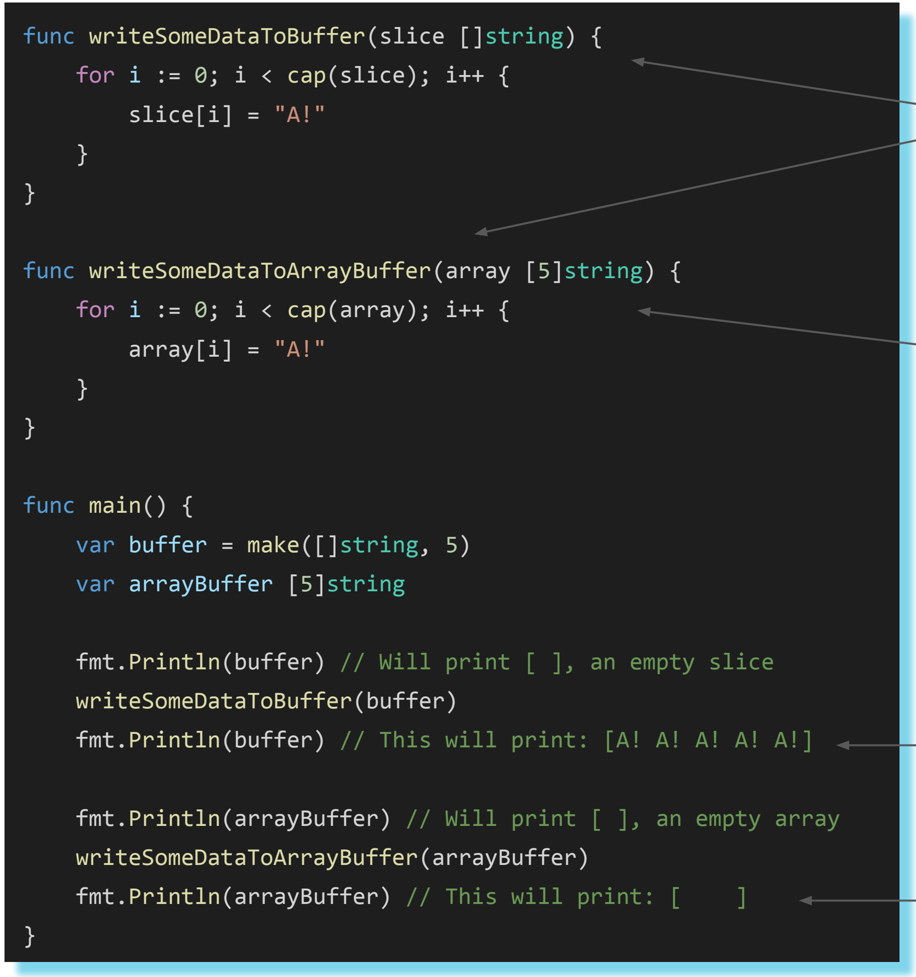
As mentioned, Go is a pass-by-value language, every time we pass a function argument, the function is actually receiving a copy of the passed data, this could mean that allocating a lot of memory in certain scenarios.

```
r = open_resource()
while r.has_data() {
        data_chunk = r.next()
        process(data_chunk)
}
close(r)
```

If r.next returns an array of data, memory will be allocated for a copy of data_chunk on every iteration. Even though memory is managed for us by the garbage collector, the memory cleanup still needs to happen and can affect performance as it runs.

Maps and Slices are internally implemented using pointers, so passing a Map or a Slice to a function as an argument creates a copy, but the internal copy pointers will be pointing at the same memory addresses, so changes to the Map or the Slice in a function will change the original map.

Slices are basically implemented as a pointer to an array, the len and the cap integers; for this reason, they work very well as buffers: if a slice is passed to a function as an argument, the function will receive a copy of the slice, but changing a slice's copy has the effect of changing the original slice, as both inner pointers will point to the same memory addresses.



```go
func writeSomeDataToBuffer(slice []string) {
    for i := 0; i < cap(slice); i++ {
        slice[i] = "A!"
    }
}

func writeSomeDataToArrayBuffer(array [5]string) {
    for i := 0; i < cap(array); i++ {
        array[i] = "A!"
    }
}

func main() {
    var buffer = make([]string, 5)
    var arrayBuffer [5]string

    fmt.Println(buffer) // Will print [ ], an empty slice
    writeSomeDataToBuffer(buffer)
    fmt.Println(buffer) // This will print: [A! A! A! A! A!]

    fmt.Println(arrayBuffer) // Will print [ ], an empty array
    writeSomeDataToArrayBuffer(arrayBuffer)
    fmt.Println(arrayBuffer) // This will print: [    ]
}
```

No function overloading in Go, so functions need to be called differently even if they have different parameters

The size is part of the array's type, we can't just pass "an array", we need to specify "an array of size 5".

The function overwrote the slice's contents

The array remained empty even though it was "overwritten"! This is because the function actually received a copy of the array, and modified that copy, the original array remained unmodified

Methods

Methods are just syntactic sugar for a function that takes a struct as an argument; however considering Go doesn't support overloading, it's a useful feature to have.

```go
type Book struct {
    Name        string
    Author      string
    LastUpdated time.Time
}

func (b Book) ToString() string {
    return fmt.Sprintf("%s written by %s", b.Name, b.Author)
}

func main() {
    b := Book{"The Dying Earth", "Jack Vance"}
    fmt.Println(b.ToString())
}
```

Unlike other languages, this is not "the" object the method will be invoked on, it will be a copy of it. This is called a "value receiver".

Since the variable b in the method signature will actually contain a copy of the object, because Go is always pass-by-value, **we cannot write a method like this and have it mutate the object**.
To have a mutator method, we need to use a pointer, pointers parameters indicate that the parameter might be changed by the method, we use it for mutability.

We need our methods to use a pointer receiver instead of a value receiver if we want to modify the receiver, or if we want our method to be able to handle a nil instance.

```go
func (b *Book) SetAuthor(author string) {
	b.Author = author
	b.LastUpdated = time.Now()
}

func main() {
	b := Book{"The Dying Earth", "Wrong author", time.Time{}}
	b.SetAuthor("Jack Vance")
	fmt.Println(b.ToString())
}
```

It doesn't matter that we have a value (so, a non pointer) type instead of a pointer type when we invoke a method, Go is smart and will convert your value type into a pointer type, like in the example above.

Don't write getters and setters in Go unless you really need to, usually to implement an interface, struct fields are accessed directly.

Methods need to be defined in the same package where the type is defined, it's not possible to write extension methods like in C#, which enable us to add functionality to types we don't own.

Calling a value-receiver method on a nil instance will cause the program to panic.

```
var b *Book
fmt.Println(b.ToString())
```

This will fail because our implementation of the ToString method has a value-receiver: func (b Book) ToString() string
We can write our code to expect and handle nil, but only if we use a pointer-receiver.
We can rewrite the function to this:

```
func (b *Book) ToString() string {
    if b == nil {
        return ""
    }

    return fmt.Sprintf("%s written by %s", b.Name, b.Author)
}
```

Methods can be assigned to variables and used as regular functions, this is called a method value:

```
myFunction := b.ToString
fmt.Println(myFunction())
```

Or you can create a function from the type, this is called a method expression:

```
bookToString := Book.ToString
fmt.Println(bookToString(b))
```

User defined types

Besides defining types based on structs, we can define them based on other types, including other user types.

```go
type Percentage int
type UserName string
type CustomerUserName UserName
```

The main use case for these is documentation, user types are not like inheritance, a Percentage is not an int, we say that it's underlying type IS an int, but they're different types. A function that takes an int as a parameter will not accept a Percentage, unless you cast it.

```go
func DoSomethingWithAnInt(param int) {
    ...
}

var percentage Percentage = 50
DoSomethingWithAnInt(int(percentage))
```

Composition (embedding)

As mentioned, there is no inheritance in Go. To extend a struct, we use embedding.

```go
type Employee struct {
    Name    string
    ID  int
}

func (e Employee) ToString() string {
    return fmt.Sprintf("Employee: %s (%s)", e.Name, e.ID)
}

type Manager struct {
    Employee
    Reports []Employee
}

func (m Manager) FindNewEmployees() []Employee {
    ...
}
```

This is not a field, Employee is embedded into the Manager type, this means that all of its fields and methods are promoted to the Manager type, as if they were defined in it. You can tell it's not a field because it doesn't have a name!

```go
m := Manager{
    Employee: Employee{
        Name: "Manager McManagerman",
        ID:   123,
    },
    Reports: []Employee{},
}

fmt.Println(m.Name)       // prints “Manager McManagerman”
fmt.Println(m.ToString()) // prints “Employee: Manager McManagerman (abc1234)
```

The Name field was defined in the Employee type, but here it is accessible and useable as if it had been originally defined in Manager. Same thing happens to the ToString method, it was promoted from the Employee type to the Manager type via embedding.

A Manager is not and Employee, a Manager contains everything defined for the Employee type, it is not the same.

```go
var e Employee = m // Compilation error
```

However it is possible to access the embedded Employee in the Manager:

```go
var e Employee = m.Employee
```

There is no dynamic dispatch-like mechanism: methods in an embedded struct have no idea they are embedded.

```go
type Inner struct {
}

type Outer struct {
    Inner
}

func (i Inner) GetType() string {
    return "Inner"
}

func (o Outer) GetType() string {
    return "Outer"
}

func (i Inner) ToString() string {
    return i.GetType()
}

func main() {
    o := Outer{
        Inner: Inner{},
    }
    fmt.Println(o.ToString()) // Prints “Inner”
}
```

Despite the GetType method existing in both types, the ToString() method only knows what is defined for the Employee type, thus it will only see the GetType that returns “Inner”

What if both the Inner and the Outer structs contain a method with the same name? How do we invoke each one?

```go
func (i Inner) ToString() string {
    return i.GetType()
}

func (o Outer) ToString() string {
    return i.GetType()
}

func main() {
    o := Outer{
        Inner: Inner{},
    }

    fmt.Println(o.ToString())       // Prints “Outer”
    fmt.Println(o.Inner.ToString()) // Prints “Inner”
}
```

Interfaces

Interfaces in Go are one of the most different aspects compared to other OO languages.

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

A type implements an interface if the type contains all the methods defined by the interface (but it can contain more), it's an **implicit** implementation rather than explicit like in Java, our types don't need anything like “implements MyInterface” or similar.

```go
type MyType struct{}

func (m MyType) String() string {
    return "Return some string"
}
```

MyType implicitly implements the interface Stringer, which is defined in the fmt standard package.

```go
type Order struct {
    OrderCode string
}

type DataRepository interface {
    GetAllOrders() []Order
}

type MockDataRepository struct{}

func (m MockDataRepository) GetAllOrders() []Order {
    return []Order{
        Order{"1000001"},
        Order{"1000002"},
    }
}

type Service struct {
    Repository DataRepository
}

func main() {
    s := Service{
        Repository: MockDataRepository{},
    }

    // do some stuff
}
```

The MockDataRepository method set contains all the functions defined by the DataRepository, therefore MockDataRepository implicitly implements the DataRepository interface, and can be used wherever that interface is required, such as in the Service type

Interfaces can embed other interfaces, like structs do.

```go
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}

type ReadCloser interface {
    Reader
    Closer
}
```

In Go it’s idiomatic to return concrete classes instead of interfaces, unlike in other OO languages, and accept interfaces as parameter.
A gotcha to keep in mind: an interface instance is nil only if both the type and the value are nil.

```go
var s *string
fmt.Println(s == nil) // prints true
var i interface{}
fmt.Println(i == nil) // prints true
i = s
fmt.Println(i == nil) // prints false
```

This is more or less equivalent to declaring a variable of type Object in C# or Java.

Function Types

Go methods can be added to any user defined type, and a function can also be a user defined type. So even functions can have their own methods, crazy!
These are used as a bridge between functions/methods/closures and interfaces

```go
type Handler interface {
    Handle(val string) (string, error)
}

func HandleUsingHandler(handler Handler, valueToHandle string) {
    result, error := handler.Handle(valueToHandle)
    fmt.Println("Handled using Handler, response was: ", result, ", and error: ", error)
}
```

Now let’s say we want to execute this function, which is probably not owned by us (so we cannot change it), and it requires a type implementing the Handler interface. If it was Java, we could pass an anonymous implementation of the interface, or use a lambda function and a closure. In Go we’d use a function type, which would be very similar to a delegate in C#.

```go
type HandlerBridge func(val string) (string, error)

func (h HandlerBridge) Handle(val string) (string, error) {
    return h(val)
}
```

We declared a type called HandlerBridge, with a backing type `func(string) (string, error)`, and a Handle method which invokes itself. It's an adapter of some sort. Any function of type `func(string) (string, error)` can now implement the interface by doing a simple type conversion (cast).

```go
func main() {
    HandleUsingHandler(HandlerBridge(func(val string) (string, error) {
        return val + " invoked from my closure!", nil
    }), "Test")
}
```

Go encourages to have many and small interfaces, so it's common to see interfaces with just one defined operation.

Errors

By convention, functions should return a value of type error as the last return value. If everything went well, the last return value should be nil, otherwise it should contain the actual error. **There is no try-catch (and no exceptions) in Go**.

```go
func divAndRemainder(numerator int, denominator int) (int, int, error) {
    if denominator == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
    return numerator / denominator, numerator % denominator, nil
}

func main() {
    numerator, remainder, err := divAndRemainder(10, 0)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    // something else here...
}
```

error is an interface

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

Therefore you can implement your own custom errors, just like you would create your own Exceptions.

There are some built-in errors called Sentinel Errors defined in the standard library.

Always return an error interface in your function definition, don't specify specific error implementations to reduce coupling.

When creating an error to return from your function, create the error on the very return line to avoid returning a non nil error when you wanted to actually return nil:

If there is an error, create it on the return line: `return nil, errors.New("Something bad happened!")`

```
var error Error
// more stuff
if somethingHappened {
    error = errors.New("Something bad happened!")
}
...
return result, error // here, even if the conditional was never triggered,
     //error would be non nil, because interface instances are not nil when they have a type.
```

Don't do this!

Panics

When a program fails unexpectedly and cannot recover from the error, it will panic. **Panic is very similar to an unhandled exception**, it will bubble up and kill your program with a stack trace.

We can have our program **panic manually**, if we decide that a condition from which our program should not continue has been met:

```
if somethingReallyTerribleHappened {
    panic("Oh no!")
}
```

There is a way to "catch" a panic and do something about it, it's called a **recover**, and **is a function called from a defer**.

```
func myFunc() {
    defer func() {
        if p := recover(); p != nil {
            doSomethingAboutWhatJustHappened(p)
        }
    }()
    // some stuff
}
```

If a panic occurs while myFunc is executing, the deferred recovery function will enter the conditional, and you can do something.

The **recover function looks a lot like a try-catch, however it's not** intended to be that. The idea of the recover function is to be able to **finish your app gracefully**, as a panic could be something like the app running out of memory from which you can certainly not move forward.

Use the convention of returning an error as the last return value of your functions instead of relying on panic-recover.

The rule is that **panics should not escape their packages**. If there is a panic in our package, we should do the recover to transform the panic into an error, so consumers of our package are not exposed to the panic.

Goroutines (concurrency!)

Concurrency in Go is different.

At its core, it uses something called **Goroutines**, which can be thought of as abstractions on top of raw Threads that simplify their usage and scheduling. They are scheduled by **Go's own scheduler**, to maximize efficiency.

To have a rough frame of reference, the Java Virtual Machine will probably go down after spawning a few tens of thousands of threads.
On the same machine, Go can run many **tens or even hundreds of millions of goroutines**.
All of this is variable machine to machine, and configuration to configuration, but gives us an idea that Go is very good at multithreading.

One of the main differences is that Java uses kernel-level OS threads, whereas Go uses user-space "threads", **it will run multiple goroutines on a single OS thread**.

To spawn a new goroutine to parallelize our workload, Go's syntax is super simple.

```go
func runInBackground() <-chan struct{} {
    done := make(chan struct{})

    go func() {
        fmt.Println("I'm running on another goroutine, probably on another thread!")

        close(done)
    }()

    return done
}

func main() {
    done := runInBackground()
    _ = <-done
}
```

The "go" keyword will spawn a new goroutine, and execute the function passed to it, in this case it's an anonymous function.

This syntax means "read from channel called done"

In this example we see how easy it is to launch a new goroutine, but we see there is something new here with the "done" variables, these are channels.

Channels

Channels are a concept that are used for goroutines, and their closest equivalent in Java or C# would be a **ConcurrentQueue**.
They are used to communicate between goroutines, to pass information around but also to signal that something has occurred.

We make a channel similarly to how to make a slice:

```
ch := make(chan int)
```

We *send* information to the channel with ->

```
ch <- 5
```

We *receive* information from the channel with <-

```
readValue := <- ch
```

We can close a channel, signalling that there is no more information to be sent through there, a closed channel can no longer be written to.

```
close(ch)
```

We can read from a channel until there is nothing else to read (which is indicated by the event of the channel being closed)

```
for value := range ch {
    ...
}
```

Quick example, a consumer producer.

```go
func getNumbers() chan int {
    ch := make(chan int)

    go func() {
        for i := 0; i < 100; i++ {
            ch <- i
        }

        close(ch)
    }()

    return ch
}

func main() {
    for val := range getNumbers() {
        fmt.Println(val)
    }
}
```

This channel is the one our function will write the numbers to

The numbers are written to the channel, it will block until someone reads them

The work is done, we need to close the channel, signalling nothing else will be written to it

While the channel is open, the for will continue trying to read from it.

Buffered and Unbuffered Channels

By default, channels are unbuffered:

- If someone reads from the channel, and there is no data, the reader will block until there is data to read.
- If someone writes to the channel, and there is already data there pending to be read, the writer will block until someone reads the existing data.

An unbuffered channel is like a concurrent blocking queue of size 1.

A buffered channel is like a concurrent blocking queue of size N, it accepts N writes to it before blocking the writer.

```
ch := make(chan int, 10)
```

For buffered channels:

- `len(ch)` will return how many data items there are in the channel.
- `cap(ch)` will return how many items it can hold.

For unbuffered, both functions return 0.

Unbuffered channels are more common.

Closing channels

When we're done using a channel, we should always close it. Closing the channel will free resources and will signal readers that nothing else will be written to it, so they can stop trying to read from it.

Writes to a closed channel will panic.

Reads from a closed channel will always return the Zero value of the channel type (for example, reading from a closed "chan int" channel will always return 0).

We can tell whether the channel has returned Zero because it's closed and not because it actually has the value 0 in it using the "**comma ok**" idiom we saw earlier.

```go
value, ok := <- ch
if !ok {
    fmt.Println("The channel is closed")
}
```

Deadlocks

Go has interesting solutions to deal with deadlocks.

First, every application consists of N goroutines (the main function will execute in it's own goroutine when we execute it), if Go detects at any point that all goroutines are deadlocked, it will terminate the application, it will not just hang like in other runtimes.

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

Second, the select structure. One of the causes of deadlocks is acquiring locks in inconsistent order.
Given two threads (or goroutines in our case), A and B.

A acquires the lock 1
B acquires the lock 2
A tries to acquire the lock 1, but blocks because the lock is retained by B
B tries to acquire the lock 2, but blocks because the lock is retained by A

A and B will be stuck in a deadlock, each one waiting for the other.

```go
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)

    go func() {
        v := 1
        ch1 <- v
        v2 := <-ch2
        fmt.Println(v, v2)
    }()

    v := 2
    ch2 <- v
    v2 := <-ch1

    fmt.Println(v, v2)
}
```

Select

The select construct is interesting, it's like a switch.

```go
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)

    go func() {
        v := 1
        ch1 <- v
        v2 := <-ch2
        fmt.Println(v, v2)
    }()
    v := 2
    var v2 int

    select {
    case ch2 <- v:
    case v2 = <-ch1:
    }
    fmt.Println(v, v2)
}
```

Try to write to ch2

Try to read from ch2

The select will **choose randomly between the cases (unlike a switch)** and will execute one of them IF it's possible, if it would be blocked by attempting to do so, it will choose another case.

In this example, it helps prevent the deadlock, because it will detect that ch2 is empty and thus would block if read from, so it instead runs the second case, unblocking the other goroutine.

Adding a default helps in the case in which no branch can be executed because all would block.
It could be used, for example, to implement a non-blocking read from a channel.

```go
func NonBlockingRead(ch <-chan int) (int, bool) {
    select {
    case v := <-ch:
        return v, true
    default:
        return 0, false
    }
}
```

We can specify in the type that we're only going to be reading from the channel, like in this case, this will help the compiler prevent accidental writes, or reads (if the channel is write only).

Channels can also be used to cancel a goroutine.

```go
func doSomethingAsync() (<-chan int, func()) {
    ch := make(chan int)
    done := make(chan struct{})
    cancel := func() {
        close(done)
    }
    go func() {
        for i := 0; i < SOME_NUMBER; i++ {
            select {
            case <-done:
                return
            case ch <- i:

            }
        }
        close(ch)
    }()
    return ch, cancel
}
```

This function is what we will use to cancel the goroutine

If someone closes the done channel, this branch will execute and finish the goroutine

Cleanup your channels

```go
func main() {
    ch, cancel := doSomethingAsync(10)
    for i := range ch {
        if i > 5 {
            break
        }
        fmt.Println(i)
    }
    cancel()
}
```

The main goroutine will execute the cancel function returned by the doSomethingAsync function. This function in turn will close the done channel. Closing the channel will cause the select structure to run the case <-done branch and finish the function.

Modules

A module is the root of a Go application.
A module contains one or more packages.
The go.mod file is the module definition, it contains the name of the module, and it contains its dependencies, this is interesting because dependency management is built into Go, so no need for Maven or Gradle.

To enable dependency management, use the command go mod init <module path>

The module path is the canonical name for your module, and it indicates where to find it, for example the module path for the Goavro library (LinkedIn's library for dealing with the binary serialization format Avro) is "github.com/linkedin/goavro/v2".

This is different than Gradle and Maven, the dependency's name also indicates where to fetch the files from, it's both weird and neat!

Packages

Packages are how we organize our Go code, they're the same as other languages with some elements of notice:

- The code that is meant to be **executed as an application** (with a main() func) needs to be in the special package "**main**", which is required to be at the root folder of our project.
- Other packages are called the same as the folders they live in, this can be overridden but it's convention so try not to break it.

When we import a package, we import all of the exported identifiers (functions, variables, types), "exported" can be translated as "public" on other languages.

```go
import (
    "fmt"
    avro "github.com/linkedin/goavro/v2"
)
```

This is the same notation as when we declare multiple variables.
In the Avro import, we're setting "avro" as an alias, otherwise the package name to be referenced in code would have been "goavro", which is the "name" portion of the full package name.

Identifiers are exported, or made public, in Go by capitalization, no need to use a keyword like exported or public.

```go
func thisWillBeAPrivateFunction() { … }

func ThisWillBeAPublicFunction() { … }
```

Testing

Tests in Go are easy!
To test a file called "calculator.go", you will add a file called "calculator_test.go" in the same folder. Test files have access to all non-exported identifiers, as if it could access internal declared members in Java.

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

To test this function, in the test file we will add a test function with the "Test_" prefix, these functions take a single `*testing.T` parameter.

```go
func Test_add(t *testing.T) {
    result := add(2, 3)
    if result := 5 {
        t.Error("Expected 5, got", result)
    }
}
```

To run tests, we use the `go test` command.

There is a special function for setting up some initial test data or structures you may need, called `TestMain`

```go
func TestMain(m *testing.M) {
    fmt.Println("Set up stuff for tests here")

    // SETUP
    // AND CALL YOUR TESTS FROM HERE
    exitVal := m.Run()

    fmt.Println("Clean up stuff after tests here")
    os.Exit(exitVal)
}

func Test_Func(t *testing.T) {
    fmt.Println("I am a test function!")
}
```

This function is called when we run go test on a package with a `TestMain` function, otherwise it will invoke the tests directly. This function is only called once, it's not called once per test.

The `*testing.T` parameter contains more helper functions to help your tests, such as creating temporary files, cleaning them up after you're done, reporting test results (success or failure).

Running go `test -cover` will include test coverage, and there's also a tool to generate HTML reports included.

Thanks to Go's implicit interfaces mechanism, writing stubs is very simple, in fact we already seen an example!

```go
type DataRepository interface {
    GetAllOrders() []Order
    GetOrder() Order
    CreateOrder(code string) int
}

type Service struct {
    Repository DataRepository
}

type MockDataRepository struct{
        DataRepository
}

func (m MockDataRepository) GetAllOrders() []Order {
    return []Order{
        Order{"1000001"},
        Order{"1000002"},
    }
}
```

Here we are **embedding** the original interface into the stub. This has the effect that the stub now has a "blank" method per each function defined in the interface. Calling one of these blank methods causes a panic. In this example we're stubbing only GetAllOrders, leaving the other functions as blank.

The End

Thank you!

References

- go.dev
- Effective Go - The Go Programming Language (golang.org)
- Visual Studio Code - Code Editing. Redefined
- (19) GopherCon 2018: Kavya Joshi - The Scheduler Saga - YouTube
- Learning Go, an idiomatic approach to Real World Go Programming, by Jon Bodner.
- Official language specification