gobook Documentation

Release 1

Big Yuuta

October 17, 2016

Contents

See the little gopher? Isn't it cute? Yes it is! He is the mascot of the Go programming language —*Go, what?* Why Go, the programming language of course!

This website is an online book that aims to introduce people to this awesome language. And if you, actually, stumbled upon it while looking for real gophers, let me still invite you to read it and learn something cool! C'mon, go-pher it!

CHAPTER 1

Welcome aboard

In this part, we will learn the very basic things about Go. The basic syntax for programs, how to declare variables, pointers and how to use basic control structures.

1.1 Preface

Go is a programming language created by a team of *very* talented folks at Google with the help of many people from the Open Source community. Its source code is open source itself and can be viewed and modified freely.

This book will try to teach you how to write some code in Go, while having fun doing so. It comes with many illustrative examples and diagrams in order to make the studied concepts easy to understand.

1.1.1 Why Go?

There are gazillions (not that many actually) of programming languages already in the wild. So one could, and rightly so, ask why yet another language? The fact is Go is easy to grok, and fun to use but also efficient and well designed.

Go is a compiled language, i.e. a language that produces machine code that can be executed without the need of an external interpreter like Python, Ruby, Perl and other scripting languages.

In fact, if we were to compare Go to something, it would be a modern C. Yes, that's a strong statement that calls for a justification. Being a C programmer myself, C has its very own dear place in my heart, a place that won't be shared by anything else. But Go earned my respect and that of many others thanks to its simplicity, efficiency and smart concepts that this book aims to demonstrate.

Go comes with some new syntax and idioms but not too many. Enough to make things evolve, but not too many to alienate people who are used to some other programming language. This balance, and -I'd dare to say- *minimalism* are what makes Go fun to learn.

Go has the efficiency of statically-typed languages, yet it provides some easy syntax and concepts that make it as *cool* as dynamic languages.

Go programs are compiled fast and run fast. Go also natively supports concurrency and communication.

1.1.2 Why this book?

This modest work is an introduction to the Go language. It is meant to be easy to understand by everyone and tries to make this learning experience fun and easy. Yes, that's quite a challenge, but... *aude aliquid dignum!* [1].

[1] 16th century Latin for "Dare something worthy"

I personally don't buy the *"learn the hard way"* method. Learning shouldn't be *hard*! Just remember that we've learnt how to speak while playing when we were younger and stupider. And Go, or any other language for that matter, is actually a lot easier than any spoken language; there are fewer words, fewer syntax rules, fewer idioms...

So here's the deal and the only condition for learning with this book: take it easy. Really, consider it as a game.

We will play with data and code, learn something new every time. There will be some diversity, from simple, or even silly things, to some very serious and smart concepts.

Also, almost -if not all- the provided programs can be tested online in the Go Playground simply by copying/pasting them there.

1.1.3 A work in progress

This is a personal initiative, I have no deadlines with any publisher. I do this for fun, and during my free time. I will strive to improve it as much as I can. I don't claim to be *perfect* —no one is!

So your help, criticism, suggestions are more than welcome. And the more you contribute, the better it will be.

Thanks for your attention, and have fun!

1.2 If you're not a programmer

This chapter is for people who have never written a single program in an imperative programming language. If you are used to languages likes C, C++, Python, Perl, Ruby or if you know what the words variable, constant, type, instruction, assignation mean you may (No! You SHOULD!) skip this section entirely.

1.2.1 How do programs work?

In the programming world there are many families of languages: functional, logic, imperative.

Go, like C, C++, Python, Perl and some others are said to be imperative languages because you write down what the processor should execute. It's an imperative way to describe tasks assigned to the computer.

An imperative program is a sequence of *instructions* that the processor executes when the program is compiled and *transformed* to a binary form that the processor understands.

Think of it as a list of steps to assemble or build something. We said *"sequence"* because the **order** of these steps generally matters a LOT! You can't paint the walls of a house when you haven't built the walls yet. See? Order matters!

1.2.2 What are variables?

When you solve math problems (mind you it won't be about math this time), you say that the variable `x` is set to the value `1` for example.

Sometimes you have to solve equations like: `2x + 1 = 5`. Solving here means finding out the value of the variable `x`.

So it's safe to say that `x` is a container that can contain different values of a given type.

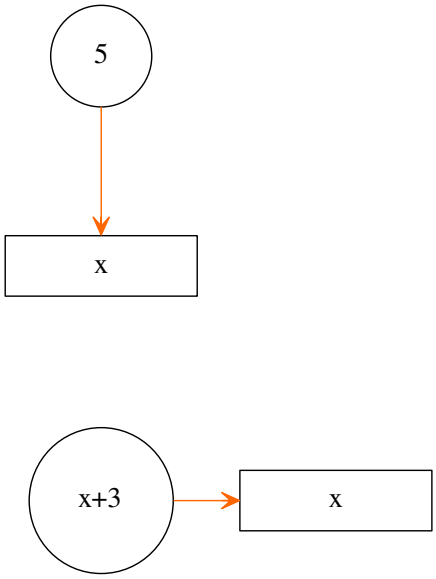
Think of them as boxes labeled with *identifiers* that store values.

1.2.3 What is assignation?

We assign a value to a variable to say that this variable is *equal* to this value from the assignation act on, until another assignation changes its value.

```
1 //assign the value 5 to the variable x
2 x = 5
3 // Now x is equal to 5. We write this like this: x == 5
4 x = x + 3 //now, we assign to x its value plus 3
5 //so now, x == 8
```

Assignation is the act of *storing* values inside variables. In the previous snippet, we assigned the value `5` to the variable whose identifier is `x` (we say simply the variable `x`) and just after, we assigned to the variable `x` its own value (`5` is what we assigned to it in the previous instruction) plus `3`. So now, x contains the value `5+3` which is `8`.



1.2.4 What is a type?

Each declared variable is of a given type. A type is a set of the values that the variable can store. In other words: A type is the set of values that we can assign to a given variable.

For example: A variable x of the type `int`eger can be used to store integer values: 0, 1, 2, ...

The phrase *"Hello world"* can be assigned to variables of type `string`.

So we actually *declare* variables of a given *type* in order to use them with values of that type.

1.2.5 What is a constant?

A constant is an identifier that is assigned a given value, which can't be changed by your program.

We *declare* a constant with a given value.

```
// Pi is a constant whose value is 3.14
const Pi = 3.14

// Let's use the constant Pi
var x float32 //we declare a variable x of type float32
x = 3 * Pi //when compiled, x == 3 * 3.14

var y float32
y = 4 * Pi //when compiled, y == 4 * 3.14
```

For example, we can declare the constant `Pi` to be equal to `3.14`. During the execution of the program, the value of `Pi` won't and can not be changed. There is no assignation of values to constants.

Constants are useful to make programs easy to read and to maintain. In effect, writing `Pi` instead of the floating point number `3.14` in many spots of your program make it easier to read.

Also, if we decide to make the ouput of our program that uses `Pi` more precise, we can change the constant's declared value once to be `3.14159` instead of the earlier `3.14` and the new value will be used all along when the program is compilled again.

1.2.6 Conclusion

Imperative programming is easy. Easy as putting values in boxes. Once you understand this very basic thing, complex (notice, I didn't say complicated) things can be seen as a composition of simpler things.

Easy and fun. Fun as playing with Lego sets :)

1.3 Konnichiwa, World!

Before starting to write Go applications like a wizard, we should start with the very basic ingredients. You just can't craft an airplane when you don't know what a wing is! So in this chapter we're going to learn the very basic syntax idioms to get our feet wet.

1.3.1 The program

This is a tradition. And we do respect some traditions [1]. The very first program one should write when learning a programming language is a small one that outputs the sentence: `Hello World`.

Ready? Go!

```
package main

import "fmt"

func main() {
```

[1] It is said that the first "hello, world" program was written by Brian Kernighan in his tutorial "Programming in C: A Tutorial" at Bell Labs in 1974. And there is a "hello, world" example in the seminal book "The C programming Language" much often refered to as "K&R" by Brian Kernighan and Dennis Ritchie.

```
6     fmt.Printf("Hello, world; καλημ ρα κόσμ or \n")
7 }
```

Output:

Hello, world; καλημ ρα κόσμ or

1.3.2 The details

The notion of packages

A Go program is constructed as a "package", which may in turn use facilities from other packages.

The statement `package <something>` (in our example: `<something>` is main) tells which package this file is part of.

In order to print on screen the text "Hello, world..." we use a function called `Printf` that comes from the `fmt` package that we imported using the statement `import "fmt"` on the 3rd line.

Note: Every standalone Go program contains a package called main and its main function, after initialization, is where execution starts. The main.main function takes no arguments and returns no value.

This is actually the same idea as in perl packages and python modules. And as you may see, the big advantage here is: modularity and reusability.

By *modularity*, I mean that you can divide your program into several pieces each one answering a specific need. And by *reusability*, I mean that packages that you write, or that already are brought to your by Go itself, can be used many times without rewriting the functionalities they provide each and every time you need them.

For now, you can just think of packages from the perspective of their *rasion d'être* and advantages. Later we will see how to create our own packages.

The main function

On line 5, we declare our main function with the keyword `func` and its body is enclosed between `{` and `}` just like in C, C++, Java and other languages.

Notice that our `main()` function takes no arguments. But we will see when we will study functions, that in Go, they can take arguments, and return no, or one, or many values!

So on line 6, we called the function `Printf` that is defined in the `fmt` packages that we imported in line 3.

The `dot notation` is what is used in Go to refer to data, or functions located in imported packages. Again, this is not a Go's invention, since this technique is used in several other languages such as Python: `<module_name>.<data_or_function>`

UTF-8 Everywhere!

Notice that the string we printed using `Printf` contains non-ASCII characters (greek and japanese). In fact, Go, natively supports UTF-8 for string *and* identifiers.

1.3.3 Conclusion

Go uses packages (like python's modules) to organize code. The function `main.main` (function main located the in main package) is the entry-point of every standalone Go program. Go uses UTF-8 for strings and identifiers and is not limited to the old ASCII character set.

1.4 The basic things

In the previous chapter, we saw that Go programs are organized using *packages* and that Go natively supports UTF-8 for strings and identifiers. In this chapter we will see how to declare and use variables and constants and the different Go built-in types.

1.4.1 How to declare a variable?

There are several ways to declare a variable in Go.

The basic form is:

```
// declare a variable named "variable_name" of type "type"
var variable_name type
```

You can declare several variables of the same type in a single line, by separating them with commas.

```
// declare variables var1, var2, and var3 all of type type
var var1, var2, var3 type
```

And you can initialize a variable when declaring it too

```
/* declare a variable named "variable_name" of type "type" and initialize it
    to value*/
    var variable_name type = value
```

You can even initialize many variables that are declared in the same statement

```
/* declare a var1, var2, var3 of type "type" and initialize them to value1,
value2, and value3 respectively*/
var var1, var2, var3 type = value1, value2, value3
```

Guess what? You can omit the type and it will be inferred from the initializers

```
/* declare and initialize var1, var2 and var3 and initialize them
respectively to value1, value2, and value3. /
var var1, var2, var3 = value1, value2, value3
```

Even shorter, inside a function body (let me repeat that: only inside a function body) you can even drop the keyword `var` and use the `:=` instead of =

```
// omit var and type, and use ':=' instead of '=' inside the function body
func test(){
    var1, var2, var3 := value1, value2, value3
}
```

Don't worry. It's actually easy. The examples with the builtin types, below, will illustrate both of these forms. Just remember that, unlike the C way, in Go the type is put at the end of the declaration and -should I repeat it?- **that the := operator can only be used inside a function body**.

1.4.2 The builtin types

Boolean

For boolean truth values, Go has the type `bool` (like the C++ one) that takes one of the values: `true` or `false`.

```
//Example snippet
var active bool //basic form
var enabled, disabled = true, false //type omitted, variables initialized
func test(){
    var available bool //general form
    valid := false //type and var omitted, and variable initialized
    available = true //normal assignation
}
```

Numeric types

For integer values, signed and unsigned, Go has `int` and `uint` both having the appropriate length for your machine (32 or 64 bits) But there's also explicit sized ints: `int8`, `int16`, `int32`, `int64` and `byte`, `uint8`, `uint16`, `uint32`, `uint64`. With `byte` being an alias for `uint8`.

For floating point values, we have `float32` and `float64`.

Wait that's not all, Go has native support for complex numbers too! In fact, you can use `complex64` for numbers with 32 bits for the real part and 32 bits for the imaginary part, and there is `complex128` for numbers with 64 bits for the real part and 64 bits for the imaginary part.

Table of numeric types

From the Go Programming Language Specification

Type	Values
uint8	the set of all unsigned 8-bit integers (0 to 255)
uint16	the set of all unsigned 16-bit integers (0 to 65535)
uint32	the set of all unsigned 32-bit integers (0 to 4294967295)
uint64	the set of all unsigned 64-bit integers (0 to 18446744073709551615)
int8	the set of all signed 8-bit integers (-128 to 127)
int16	the set of all signed 16-bit integers (-32768 to 32767)
int32	the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64	the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
float32	the set of all IEEE-754 32-bit floating-point numbers
float64	the set of all IEEE-754 64-bit floating-point numbers
complex64	the set of all complex numbers with float32 real and imaginary parts
complex128	the set of all complex numbers with float64 real and imaginary parts
byte familiar alias for uint8	

```
//Example snippet
var i int32 //basic form with a int32
var x, y, z = 1, 2, 3 //type omitted, variables initialized
func test(){
```

```
    var pi float32 //basic form
    one, two, three := 1, 2, 3 //type and var omitted, variables initialized
    c := 10+3i // a complex number, type infered and keyword 'var' omitted.
    pi = 3.14 // normal assignation
}
```

Strings

As seen in the previous chapter, strings are in UTF-8 and they are enclosed between two double quotes (") and their type is -you bet!- `string`.

```
//Example snippet
var french_hello string //basic form
var empty_string string = "" // here empty_string (like french_hello) equals ""
func test(){
    no, yes, maybe := "no", "yes", "maybe" //var and type omitted
    japanese_hello := "Ohaiou"  //type inferred, var keyword omitted
    french_hello = "Bonjour" //normal assignation
}
```

1.4.3 Constants

In Go, constants are -uh- constant values created at compile time, and they can be: numbers, boolean or strings.

The syntax to declare a constant is:

```
const constant_name = value
```

Some examples:

```
//example snippet
const i = 100
const pi = 3.14
const prefix = "go_"
```

1.4.4 Facilities

Grouping declarations

Multiple `var`, `const` and `import` declarations can be grouped using parenthesis.

So instead of writing:

```
//example snippet
import "fmt"
import "os"

const i = 100
const pi = 3.14
const prefix = "go_"

var i int
var pi = float32
var prefix string
```

You can write:

```
//example snippet with grouping
import (
    "fmt"
    "os"
)

const (
    i = 100
    pi = 3.14
    prefix = "go_"
)

var (
    i int
    pi = float32
    prefix string
)
```

Of course, you group consts with consts, vars with vars and imports with imports but you can not mix them in the same group!

iota and enumerations

Go provides the keyword `iota` that can be used when declaring enumerated constants, This keyword yelds an incremented value by 1, starting from 0, each time it is used.

Example:

```
//example snippet
const (
    x = iota //x == 0
    y = iota //y == 1
    z = iota //z == 2
    w // implicitely w == iota, therefore: w == 3
)
```

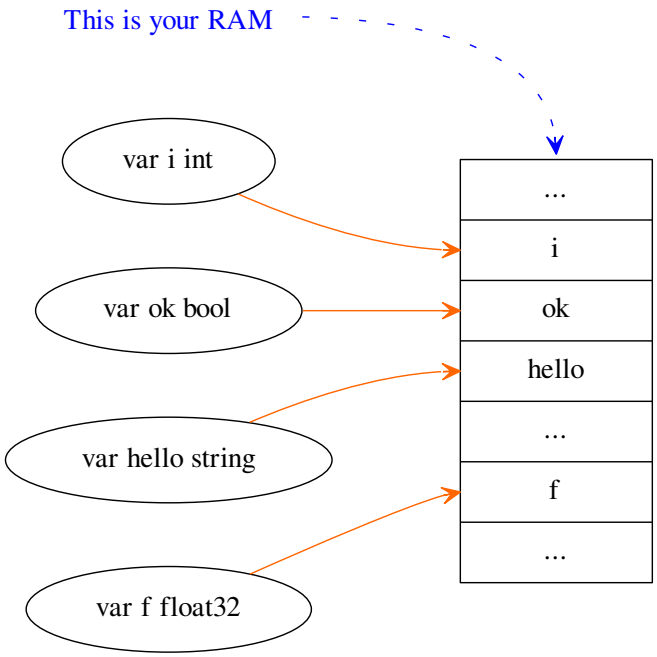
Well, that's it for this chapter. I told you, it won't be hard. In fact, Go eases variable declarations a lot. You'd almost feel like coding with a scripting language like python –and it's even better.

1.5 Pointers and memory

Most people think of pointers as an *advanced* topic. And honestly, the more you think of them as advanced, the more you'll fear them and the less you'll understand such a simple thing. You are not an idiot, let me prove that for you.

When you declare a variable in your program, you know that this variable occupies a place in your memory (RAM), right? Because when you access it, when you change its value, it must exist somewhere!

And this *somewhere* is the variable's *address* in memory.



Now, when you buy your RAM, it's nowadays sold in GB (Giga Bytes), that is how many *bytes* are available to be used by your program and its variables. And you guess, that the string "Hello, world" doesn't occupy the same amount of RAM as a simple integer.

For example, the variable `i` can occupy 4 bytes of your RAM starting from the 100th byte to the 103th, and the string `"Hello world"` can occupy more ram starting from the 105th byte to the 116th byte.

The first byte's number of the space occupied by the variable is called its address.

And we use the `&` operator to find out the address of a given variable.

```
package main

import "fmt"

var (
    i int = 9
    hello string = "Hello world"
    pi float32 = 3.14
    c complex64 = 3+5i
)

func main() {
    fmt.Println("Hexadecimal address of i is: ", &i)
    fmt.Println("Hexadecimal address of hello is: ", &hello)
    fmt.Println("Hexadecimal address of pi is: ", &pi)
    fmt.Println("Hexadecimal address of c is: ", &c)
```

```
}
```

Output:

Hexadecimal address of i is: 0x4b8018
Hexadecimal address of hello is: 0x4b8108
Hexadecimal address of pi is: 0x4b801c
Hexadecimal address of c is: 0x4b80b8

You see? Each variable has its own address, and you can guess that the addresses of variables depend on the amount of RAM occupied by some other variables.

The variables that can *store* other variables' addresses are called *pointers* to these variables.

1.5.1 How to declare a pointer?

For any given type `T`, there is an associated type `*T` for pointers to variables of type `T`.

Example:

```
var i int
var hello string
var p *int //p is of type *int, so p is a pointer to variables of type int

//we can assign to p the address of i like this:
p = &i //now p points to i (i.e. p stores the address of i
hello_ptr := &hello //hello_ptr is a pointer variable of type *string and it points hello
```

1.5.2 How to access the value of a pointed-to variable?

If `Ptr` is a pointer to `Var` then `Ptr == &Var`. And `*Ptr == Var`.

In other words: you precede a variable with the `&` operator to obtain its address (a pointer to it), and you precede a pointer by the `*` operator to obtain the value of the variable it points to. And this is called: *dereferencing*.

Remember

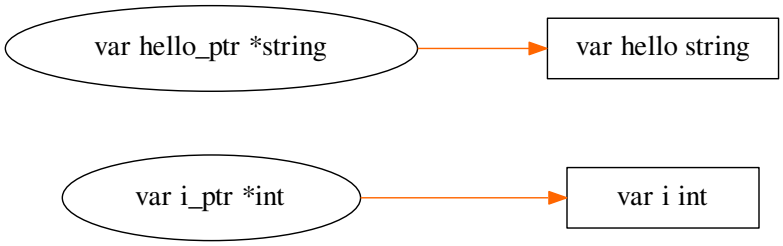
- `&` is called the address operator.
- `*` is called the dereferencing operator.

```
package main
import "fmt"

func main() {
    hello := "Hello, mina-san!"
    //declare a hello_ptr pointer variable to strings
    var hello_ptr *string
    // make it point our hello variable. i.e. assign its address to it
    hello_ptr = &hello
    // and int variable and a pointer to it
    i := 6
```

```
12      i_ptr := &i //i_ptr is of type *int
13
14      fmt.Println("The string hello is: ", hello)
15      fmt.Println("The string pointed to by hello_ptr is: ", *hello_ptr)
16      fmt.Println("The value of i is: ", i)
17      fmt.Println("The value pointed to by i_ptr is: ", *i_ptr)
18  }
```

Output:

The string hello is: Hello, mina-san!
The string pointed to by hello_ptr is: Hello, mina-san!
The value of i is: 6
The value pointed to by i_ptr is: 6



And that is all! Pointers are variables that store other variables addresses, and if a variable is of type `int`, its pointer will be of type `*int`, if it is of type `float32`, its pointer is of type `*float32` and so on...

1.5.3 Why do we need pointers?

Since we can access variables, assign values to them using only their names, one might wonder why and how pointers are useful. This is a very legitimate question, and you'll see the answer is quite simple.

Suppose that your program as it runs needs some to store some results in some new variables not already declared. How would it do that? You'll say: I'll prepare some variables just in case. But wait, what if the number of new variables differs on each execution of your program?

The answer is *runtime allocation*: your program will *allocate* some memory to store some data while it is running. Go has a built-in allocation function called `new` which allocates exactly the amount of memory required to store a variable of a given type, and after it allocates the memory, it returns a pointer.

You use it like this: `new(Type)` where `Type` is the type of the variable you want to use.

Here's an example to explain the `new` function.

```go
package main
import "fmt"

func main() {
    sum := 0
    var  double_sum *int //a pointer to int
    for i:=0; i<10; i++{
        sum += i
    }
    double_sum = new(int) //allocate memory for an int and make double_sum point to it
    *double_sum = sum*2 //use the allocated memory, by dereferencing double_sum
    fmt.Println("The sum of numbers from 0 to 10 is: ", sum)
    fmt.Println("The double of this sum is: ", *double_sum)
}
```

The sum of numbers from 0 to 10 is: 45
The double of this sum is: 90

Another good reason, that we will see when we will study *functions* is that passing parameters by reference can be useful in many cases. But that's a story for another day. Until then, I hope that you have assimilated the notion of pointers, but don't worry, we will work with them gradually in the next chapters, and you'll see that the fear some people have about them is unjustified.

1.6 Basic Control flow

In the previous chapter, we learned how to declare variables and constants. We also saw the different basic, builtin types that come with Go. In this chapter, we will see the basic control constructions, or how to *reason about* this data. But before that, let's talk about comments and semicolons.

1.6.1 Comments

You probably noticed in the example snippets given in previous chapters, some comments. Didn't you? Go has the same commenting conventions as in C++. That is:

- Single line comments: Everything from `//` until the end of the line is a comment.
- bloc comments: Everything betwwen `/*` and `*/` is a comment.

```go
//Single line comment, this is ignored by the compiler.
/* Everything from the beginning of this line until the keyword 'var' on
line 4 is a comment and ignored by the compiler. */
var(
    integer int
    pi = float32
    prefix string
)
```

1.6.2 Semicolons

If you've programmed with C, or C++, or Pascal, you might wonder why, in the previous examples, there weren't any semicolons. In fact, you don't need them in Go. Go automatically inserts semicolons on every line end that looks like

the end of a statement.

And this makes the code a lot cleaner, and easy on the eyes.

The only place you will typically see semicolons is separating the clauses of for loops and the like; they are not necessary after every statement.

Note that you can use semicolons to separate several statements written on a single line.

The one surprise is that it's important to put the opening brace of a construct such as an if statement on the same line as the if; if you don't, there are situations that may not compile or may give the wrong result [1]

Ok let's begin, now.

1.6.3 The if statement

Perhaps the most well-known statement type in imperative programming. And it can be summarized like this: if condition met, do this, else do that.

In Go, you don't need parenthesis for the condition.

```
if x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}
```

You can also have a leading initial short statement before the condition too.

```
// Compute the value of x, and then compare it to 10.
if x := computed_value(); x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}
```

You can combine multiple if/else statements.

```
if integer == 3 {
    fmt.Println("The integer is equal to 3")
} else if integer < 3 {
    fmt.Println("The integer is less than 3")
} else {
    fmt.Println("The integer is greater than 3")
}
```

1.6.4 The for statement

The `for` statement is used to iterate and loop. Its general syntax is:

```
for expression1; expression2; expression3{
    ...
}
```

Gramatically, `expression1`, `expression2`, and `expression3` are expressions, where `expression1`, and `expression3` are used for assignements or function calls (`expression1` before starting the loop, and

[1] http://golang.org/doc/go_tutorial.html#tmp_33

`expression3` at the end of every iteration) and `expression2` is used to determine whether to continue or stop the iterations.

An example would be better than the previous paragraph, right? Here we go!

```
1  package main
2  import "fmt"
3
4  func main(){
5      sum := 0;
6      for index:=0; index < 10 ; index++ {
7          sum += index
8      }
9      fmt.Println("sum is equal to ", sum)
10 }
```

In the code above we initalize our variable `sum` to 0. The `for` loop by is begun by initializing the variable `index` to 0 (`index:=0`).

Next the `for` loop's body is executed (`sum += i`) *while* the condition `index < 10` is true.

At the end of each iteration the `for` loop will execute the `index++` expression (eg. increments `index`).

You might guess that the output from the previous program would be:

```
sum is equal to 45
```

And you would be correct! Because the sum is 0+1+2+3+4+5+6+7+8+9 which equals 45.

Question:

> Is it possible to combine lines 5 and 6 into a single one, in the previous program? How?

Actually `expression1`, `expression2`, and `expression3` are all optional. This means you can omit, one, two or all of them in a `for` loop. If you omit `expression1` or `expression3` it means they simply won't be executed. If you omit `expression2` that'd mean that it is always true and that the loop will iterate forever – unless a `break` statement is encountered.

Since these expressions may be omitted, you can have something like:

```
1  //expression1 and expression3 are omitted here
2  sum := 1
3  for ; sum < 1000; {
4      sum += sum
5  }
```

Which can be simplified by removing the semicolons:

```
1  //expression1 and expression3 are omitted here, and semicolons gone.
2  sum := 1
3  for sum < 1000 {
4      sum += sum
5  }
```

Or when even expression2 is omitted also, we can have something like:

```
1  //infinite loop, no semicolons at all.
2  for {
3      fmt.Println("I loop for ever!")
4  }
```

1.6.5 break and continue

With `break` you can stop loops early. i.e. finish the loop even if the condition expressed by `expression2` is still true.

```
1 for index := 10; index > 0; index-- {
2     if index < 5{
3         break
4     }
5     fmt.Println(index)
6 }
```

The previous snippet says: loop from 10 to 0 printing the numbers (line 6); but if i<5 then break (stop) the loop! Hence, the program will print the numbers: 10, 9, 8, 7, 6, 5.

`continue`, on the other hand will just break the current iteration and *skips* to the next one.

```
1 for index := 10; index > 0; index-- {
2     if index == 5 {
3         continue
4     }
5     fmt.Println(index)
6 }
```

Here the program will print all the numbers from 10 down to 1, except 5! Because on line 3, the condition `if index == 5` will be `true` so `continue` will be executed and the `fmt.Println(index) (for index == 5)` won't be run.

1.6.6 The switch statement

Sometimes, you'll need to write a complicated chain of `if`/`else` tests. And the code becomes ugly and hard to read and maintain. The `switch` statement, makes it look nicer and easier to read.

Go's switch is more flexible than its C equivalent, because the case expression need not to be constants or even integers.

The general form of a switch statement is

```
1  // General form of a switch statement:
2  switch sExpr {
3      case expr1:
4          some instructions
5      case expr2:
6          some other instructions
7      case expr3:
8          some other instructions
9      default:
10         other code
11 }
```

The type of `sExpr` and the expressions `expr1`, `expr2`, `expr3`... type should match. i.e. if `var` is of type `int` the cases expressions should be of type `int` also!

Of course, you can have as many cases you want.

Multiple match expressions can be used, each separated by commas in the case statement.

If there is no `sExpr` at all, then by default it's *bool* and so should be the cases expressions.

If more than one case statements match, then the first in lexical order is executed.

The *default* case is optional and can be anywhere within the switch block, not necessarily the last one.

Unlike certain other languages, each case block is independent and code does not "fall through" (each of the case code blocks are like independent if-else-if code blocks. There is a fallthrough statement that you can explicitly use to obtain fall through behavior if desired.)

Some switch examples:

```
1  // Simple switch statement example:
2  i := 10
3  switch i {
4      case 1:
5          fmt.Println("i is equal to 1")
6      case 2, 3, 4:
7          fmt.Println("i is equal to 2, 3 or 4")
8      case 10:
9          fmt.Println("i is equal to 10")
10     default:
11         fmt.Println("All I know is that i is an integer")
12 }
```

In this snippet, we initialized `index` to 10, but in practice, you should think of `index` as a computed value (result of a function or some other calculus before the switch statement)

Notice that in line 6, we grouped some expressions (2, 3, and 4) in a single case statement.

```
1  // Switch example without sExpr
2  index := 10
3  switch {
4      case index < 10:
5          fmt.Println("The index is less than 10")
6      case index > 10, index < 0:
7          fmt.Println("The index is either bigger than 10 or less than 0")
8      case index == 10:
9          fmt.Println("The index is equal to 10")
10     default:
11         fmt.Println("This won't be printed anyway")
12 }
```

Now, in this example, we omitted `sExpr` of the general form. So the cases expressions should be of type *bool*. And so they are! (They're comparisons that return either `true` or `false`)

Question:

Can you tell why the `default` case on line 10 will never be reached?

```
1  //switch example with fallthrough
2  integer := 6
3  switch integer {
4  case 4:
5      fmt.Println("The integer was <= 4")
6      fallthrough
7  case 5:
8      fmt.Println("The integer was <= 5")
9      fallthrough
10 case 6:
11     fmt.Println("The integer was <= 6")
12     fallthrough
13 case 7:
14     fmt.Println("The integer was <= 7")
15     fallthrough
```

```
16 case 8:
17     fmt.Println("The integer was <= 8")
18     fallthrough
19 default:
20     fmt.Println("default case")
21 }
```

In this example, the case on line 10 matches, but since there is `fallthrough`, the code for `case 7:` and so on will also be executed (just like a C switch that doesn't use the `break` keyword).

CHAPTER 2

Let's combine stuff

Now, it's time to play like children with Legos. Technical Legos. "Tegos"? ;) We use primitive data types and basic control structures, that we saw in the previous part, to build composite types and functions.

2.1 Functions, why and how

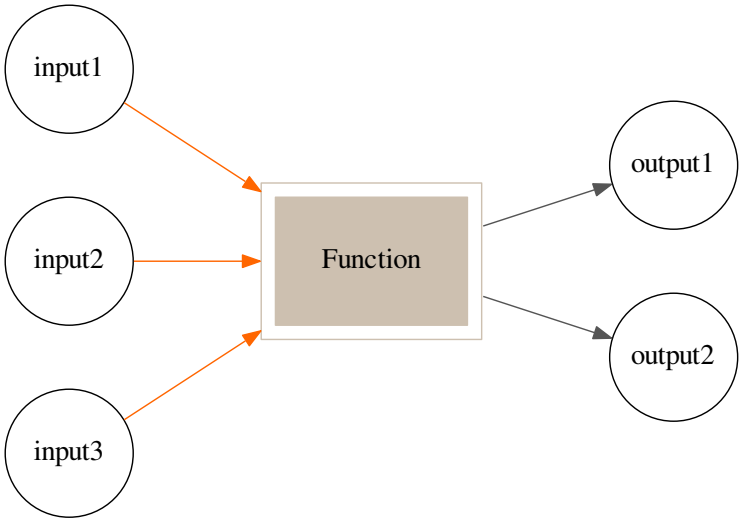
In the previous two chapters, we saw the basic data types, and the basic control structures. They're basic in the sense that they are useful for some very basic needs. And as the needs, or the problem to solve with a program become complex, you'll feel the the need for complex data, and control structures.

2.1.1 The problem

Let's say that we have to write a program to retrieve the maximum value of two integers A and B. That's easy to do with a simple *if* statement, right?

```
1 //compare A and B and say wich is bigger
2 if A > B {
3     fmt.Print("The max is A")
4 } else {
5     fmt.Print("The max is B")
6 }
```

Now, imagine that we have to make such a comparison many times in our program. Writing the previous snippet every time becomes a real chore, and error-prone. This is why we need to *encapsulate* this snippet in a code block, give it a name and then call this code block every time we need to know the max of two numbers. This code block is what we call a function.

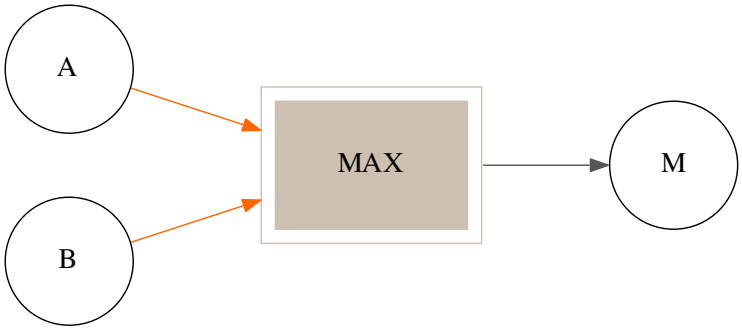


A *function* is a code block that takes some items and returns some results.

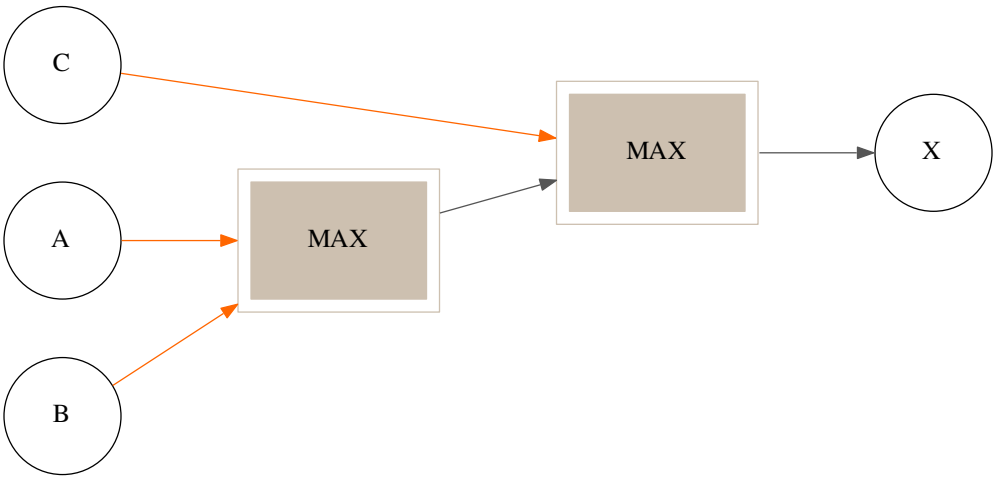
If you've ever done some electronics, you'd see that functions really look like electronic circuits or chips. Need to amplify a signal? Take an amplifier chipset. Need to filter? Use a filter chipset and so on...

Your job as a software developer will be much easier, and far more fun once you start to think of a complex project as a set of modules for which the input is from the output of another module. The complexity of the solution will be reduced, and the debugging will be made easier also.

Let's see how to apply this sort of design to the following problem: suppose we have a function MAX(A, B) which takes two integers {A, B} and returns a single integer output M which is the larger value of both A and B:



By reusing the function `MAX` we can compute the maximum value of three values: A, B and C as follows:



This is a very simple example actually. In practice you probably should write a `MAX3` function that takes 3 input integers and returns a single integer, instead of reusing the function `MAX`.

Of course, you can *inject* the output of a function into another if and only if the types match.

2.1.2 How to do this in Go?

The general syntax of a function is:

```
func funcname(input1 type1, input2 type2) (output1 type1, output2 type2) {
    //some code and processing here
    ...
    //return the results of the function
    return value1, value2
}
```

The details:

- The keyword `func` is used to declare a function of name *funcname*.
- A function may take as many input parameters as required. Each one followed by its type and all separated by commas.
- A function may return many result values.
- In the previous snippet, `result1` and `result2` are called *named result parameters*, if you don't want to name the return parameters, you can instead specify only the types, separated with commas.
- If your function returns only one output value, you may omit the parenthesis arround the output declaration portion.
- If your function doesn't return any value, you may omit it entirely.

Some examples to better understand these rules:

A simple Max function

```
package main
import "fmt"

//return the maximum between two int a, and b.
func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

func main() {
    x := 3
    y := 4
    z := 5

    max_xy := max(x, y) //calling max(x, y)
    max_xz := max(x, z) //calling max(x, z)

    fmt.Printf("max(%d, %d) = %d\n", x, y, max_xy)
    fmt.Printf("max(%d, %d) = %d\n", x, z, max_xz)
    fmt.Printf("max(%d, %d) = %d\n", y, z, max(y,z)) //just call it here
}
```

Output:

max(3, 4) = 4
max(3, 5) = 5
max(4, 5) = 5

Our function *MAX* takes two input parameters A and B, and returns a single `int`. Notice how we grouped A and B's types. We could have written: MAX(A int, B int), however, it is shorter this way.

Notice also that we prefered not to name our output value. We instead specified the output type (`int`).

A function with two ouput values

```
package main
import "fmt"

//return A+B and A*B in a single shot
func SumAndProduct(A, B int) (int, int) {
    return A+B, A*B
}

func main() {
    x := 3
    y := 4

    xPLUSy, xTIMESy := SumAndProduct(x, y)

    fmt.Printf("%d + %d = %d\n", x, y, xPLUSy)
    fmt.Printf("%d * %d = %d\n", x, y, xTIMESy)
}
```

Output:

3 + 4 = 7
3 * 4 = 12

A function with a result variable

```
package main

//look how we grouped the import of packages fmt and math
import (
    "fmt"
    "math"
)

//A function that returns a bool that is set to true of Sqrt is possible
//and false when not. And the actual square root of a float64
func MySqrt(f float64) (squareroot float64, ok bool) {
    if f > 0 {
        squareroot, ok = math.Sqrt(f), true
    } else {
        squareroot, ok = 0, false
    }
    return squareroot, ok
}

func main() {
    for index := -2.0; index <= 10; index++ {
        squareroot, possible := MySqrt(index)
        if possible {
```

```
            fmt.Printf("The square root of %f is %f\n", index, squareroot)
        } else {
            fmt.Printf("Sorry, no square root for %f\n", index)
        }
    }
}
```

Outputs:

Sorry, no square root for -2.000000
Sorry, no square root for -1.000000
Sorry, no square root for 0.000000
The square root of 1.000000 is 1.000000
The square root of 2.000000 is 1.414214
The square root of 3.000000 is 1.732051
The square root of 4.000000 is 2.000000
The square root of 5.000000 is 2.236068
The square root of 6.000000 is 2.449490
The square root of 7.000000 is 2.645751
The square root of 8.000000 is 2.828427
The square root of 9.000000 is 3.000000
The square root of 10.000000 is 3.162278

Here, we *import* the package "`math`" in order to use the `Sqrt` (Square root) function it provides. We, then, write our own function `MySqrt` which returns two values: the first one is the actual sqaure root of the input parameter `f` and the second one is a boolean (indicates whether a square root is possible or not)

Notice how we use the parameters `s` and `ok` as actual variables in the function's body.

Since result variables are initialized to "zero" (0, 0.00, false...) according to its type, we can rewrite the previous example as follows:

```
import "math"

//return A+B and A*B in a single shot
func MySqrt(floater float64) (squareroot float64, ok bool){
    if floater > 0 {
        squareroot, ok = math.Sqrt(f), true
    }
    return squareroot, ok
}
```

The empty return

When using named output variables, if the function executes a return statement with no arguments, the current values of the result parameters are used as the returned values.

Thus, we can rewrite the previous example as follow:

```
import "math"

//return A+B and A*B in a single shot
```

```
4  func MySqrt(floater float64) (squareroot float64, ok bool) {
5      if floater > 0 {
6          squareroot, ok = math.Sqrt(f), true
7      }
8      return // Omitting the output named variables, but keeping the "return".
9  }
```

2.1.3 Parameters by value, and by reference

When passing an input parameter to a function, it actually recieves a *copy* of this parameter. So, if the function changes the value of this parameter, the original variable's value won't be changed, because the function works on a *copy* of the original variable's value.

An example to verify the previous paragraph:

```
1  package main
2  import "fmt"
3
4  //simple function that returns 1 + its input parameter
5  func add1(a int) int {
6      a = a+1 // we change the value of a, by adding 1 to it
7      return a //return the new value
8  }
9
10 func main() {
11     x := 3
12
13     fmt.Println("x = ", x) // Should print "x = 3"
14
15     x1 := add1(x) //calling add1(x)
16
17     fmt.Println("x+1 = ", x1) // Should print "x+1 = 4"
18     fmt.Println("x = ", x) // Will print "x = 3"
19 }
```

x = 3
x+1 = 4
x = 3

You see? The value of `x` wasn't changed by the call of the function `add1` even though we had an explicit `a = a+1` instruction on line 6.

The reason is simple: when we called the function `add1`, it recieved a *copy* of the variable `x` and not `x` itself, hence it changed the copy's value, not the value of `x` itself.

I know the question you have in mind: *"What if I wanted the value of x to be changed by calling the function?"*

And here comes the utility of pointers: Instead of writing a function that has an `int` input parameter, we should give it a pointer to the variable we want to change! This way, the function has *access* to the original variable and it can change it.

Let's try this.

```
1  package main
2  import "fmt"
```

```
3
4  //simple function that returns 1 + its input parameter
5  func add1(a *int) int { // Notice that we give it a pointer to an int!
6      *a = *a+1 // We dereference and change the value pointed by a
7      return *a // Return the new value
8  }
9
10 func main() {
11     x := 3
12
13     fmt.Println("x = ", x) // Will print "x = 3"
14
15     x1 := add1(&x) // Calling add1(&x) by passing the address of x to it
16
17     fmt.Println("x+1 = ", x1) // Will print "x+1 = 4"
18     fmt.Println("x = ", x) // Will print "x = 4"
19 }
```

x = 3
x+1 = 4
x = 4

Now, we have changed the value of `x`!

How is passing a reference to functions is useful? You may ask.

- The first reason is that passing a reference makes function *cooperation* on a single variable possible. In other words, if we want to apply several functions on a given variable, they will all be able to change it.
- A pointer is cheap. Cheap in terms of memory usage. We can have functions that operate on a big data value for example. Copying this data will, of course, require memory on each call as it is copied for use by the function. In this way, a pointer takes much less memory. Yo! Remember, it's just an address! :)

2.1.4 Function's signatures

Like with people, names don't matter that much. Do they? Well, yes, let's face it, they do matter, but what matters most is what they do, what they need, what they produce (Yeah you! Get back to work!).

In our previous example, we could have called our function **add_one** instead of **add1**, and that would still work as long as we call it by its name. What really matters in our function is:

1. Its input parameters: how much? which types?
2. Its output parameters: how much, which type?
3. Its body code: What does the function do?

We can rewrite the function's body to work differently, and the main program would continue to compile and run without problems. But we can't change the input and output parameters in the function's declaration and still use its old parameters in the main program.

In other words, of the 3 elements we listed above, what matters even more is: what input parameters does the function expect, and what output parameters does it return.

These two elements are what we call a *function signature*, for which we use this formalism:

`func (input1 type1 [, input2 type2 [, ...]]) (output1 OutputType1 [, output2 OutputType2 [,...]])`

Or optionally with the function's name:

`func function_name (input1 type1 [, input2 type2 [, ...]]) (output1 OutputType1 [, output2 OutputType2 [,...]])`

Examples of signatures

```
1  //The signature of a function that takes an int and returns and int
2  func (int x) x int
3
4  //takes two float and returns a bool
5  func (float32, float32) bool
6
7  // Takes a string returns nothing
8  func (string)
9
10 // Takes nothing returns nothing
11 func()
```

2.2 Basic composite types

In the previous chapter, we saw how to combine the traditional control flow idioms seen in chapter 3 to create blocks of reusable code, called *functions*. In this chapter, we'll see how to combine the basic data types seen in *chapter 2*, in order to create complex data types.

2.2.1 The problem

Go comes with some *primitive* types: ints, floats, complex numbers, booleans, and strings. Fair enough, but you'll notice as you start to write complex programs that you'll need more advanced forms of data.

Let's begin with an example.

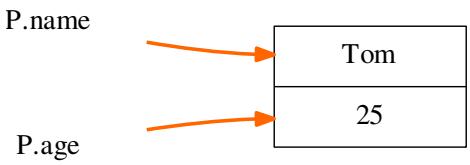
In the previous chapter, we wrote the `MAX(A, B int)` function to find out the bigger value between two integers A, and B. Now, suppose that A, and B are actually the ages of persons whose names are respectively Tom and Bob. We need our program to output the name of the older person, and by how many years he is older.

Sure, we still can use the `MAX(A, B int)` function by giving it as input parameters, the ages of Tom and Bob. But it'd be nicer to write a function `Older` for **people**. And not for integers.

People! I said people! Thus the need to *create* a custom type to *represent* people, and this type will be used for the input parameters of the `Older` function we'd like to write.

2.2.2 Structs

In Go, as in C and other languages, we can declare new types that act as *containers* for attributes or fields of some other types. For example, we can create a custom type called `person` to represent the attributes of a person. In our example these attributes will be the person's *name* **and** *age*.



A type like this is called a `struct`, which is short for *structure*.

How to declare a struct in Go?

```
type person struct {
    name string // Field name of type string.
    age int // Field age of type int.
}
```

See? It's easy. A person `struct` is a container of two fields:

- The person's name: a field of type `string`.
- The person's age: a field of type `int`.

To declare a `person` variable we do exactly as we learned in chapter 2 for basic variable types.

Thus, writing things like:

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

var P person // P will be a variable of type person. How quaint!
```

We access and assign a struct's fields (attributes) with the *dot nottation*. That is, if `P` is a variable of type `person`, then we access its fields like this:

```
P.name = "Tom" // Assign "Tom" to P's name field.
P.age = 25 // Set P's age to 25 (an int).
fmt.Println("The person's name is %s", P.name) // Access P's name field.
```

There's even two short-form assignment statement syntaxes:

1. By providing the values for each (and every) field in order:

```
//declare a variable P of type person and initialize its fields
//name to "Tom" and age to 25.
P := person{"Tom", 25}
```

2. By using field:value initializers for any number of fields in any order:

```
//declare a variable P of type person and initialize its fields
//name to "Tom" and age to 25.
//Order, here, doesn't count since we specify the fields.
P := person{age:24, name:"Tom"}
```

Good. Let's write our `Older` function that takes two input parameters of type `person`, and returns the older one *and* the difference in their ages.

```
package main
import "fmt"

// We declare our new type
type person struct {
    name string
    age int
}

// Return the older person of p1 and p2
//   and the difference in their ages.
func Older(p1, p2 person) (person, int) {
    if p1.age>p2.age {  // Compare p1 and p2's ages
        return p1, p1.age-p2.age
    }
    return p2, p2.age-p1.age
}

func main() {
    var tom person

    tom.name, tom.age = "Tom", 18

    // Look how to declare and initialize easily.
    bob := person{age:25, name:"Bob"} //specify the fields and their values
    paul := person("Paul", 43) //specify values of fields in their order

    tb_Older, tb_diff := Older(tom, bob)
    tp_Older, tp_diff := Older(tom, paul)
    bp_Older, bp_diff := Older(bob, paul)

    fmt.Printf("Of %s and %s, %s is older by %d years\n",
                tom.name, bob.name, tb_Older.name, tb_diff)

    fmt.Printf("Of %s and %s, %s is older by %d years\n",
                tom.name, paul.name, tp_Older.name, tp_diff)

    fmt.Printf("Of %s and %s, %s is older by %d years\n",
                bob.name, paul.name, bp_Older.name, bp_diff)
}
```

Output:

Of Tom and Bob, Bob is older by 7 years
Of Tom and Paul, Paul is older by 25 years
Of Bob and Paul, Paul is older by 18 years

And that's about it! `struct` types are handy, easy to declare and to use!

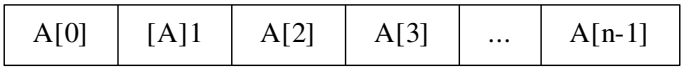
Ready for another problem? Now, suppose that we'd like to retrieve the older of, not 2, but 10 persons! Would you write a `Older10` function that would take 10 input `persons`? Sure you can! But, seriously, that would be one ugly, very ugly function! Don't write it, just picture it in your brain. Still not convinced? Ok, imagine a `Older100` function with **100** input parameters! Convinced, now?

Ok, let's see how to solve these kinds of problems using arrays.

2.2.3 Arrays

An array of size **n** is a *block* of **n** *consecutive* objects of the same **type**. That is a finite *set* of indexed objects, where you can enumerate them: *object number 1*, *object number 2*, etc...

In fact, in Go as in C and other languages, the indexing starts from 0, so you actually say: *object number 0*, *object number 1*... *object number n-1*



Here's how to declare an array of 10 `int`s, for example:

```
//declare an array A of 10 ints
var A [10]int
```

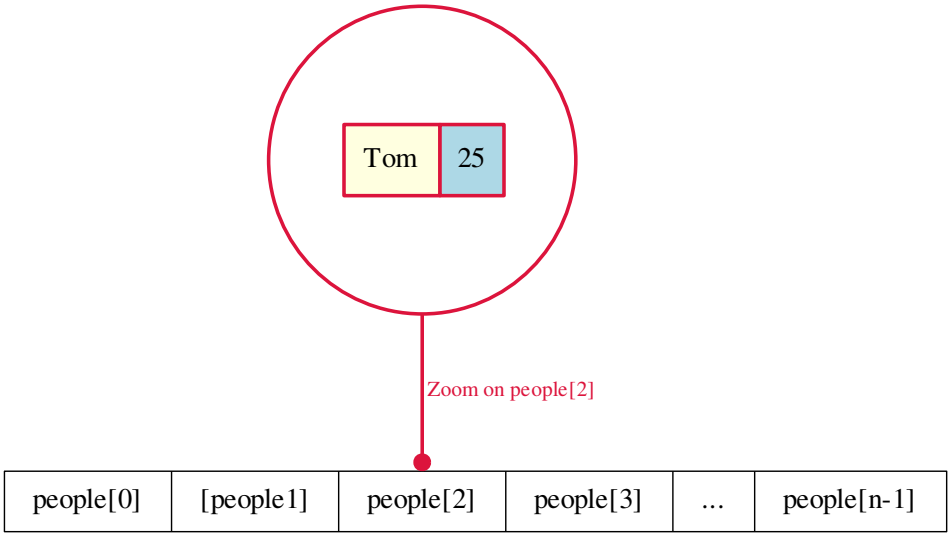
And here's how to declare an array of 10 `persons`:

```
//declare an array A of 10 persons
var people [10]person
```

We can access the x [th] element of an array `A`, with the syntax: **A[x]**. Specifically we say: `A[0]`, `A[1]`, ..., `A[n]`.

For example, the name of the 3 [rd] person is:

```
// Declare an array A of 10 persons:
var people [10]person

// Remember indices start from 0!
third_name := people[2].name //use the dot to access the third 'name'.

// Or more verbosely:
third_person := people[2]
thirdname := third_person.name
```



The idea now is to write a function `Older10` that takes an array of 10 `persons` as its input, and returns the older `person` in this array.

```
package main
import "fmt"

//Our struct
type person struct {
    name string
    age int
}

//return the older person in a group of 10 persons
func Older10(people [10]person) person {
    older := people[0] // The first one is the older for now.
    // Loop through the array and check if we could find an older person.
    for index := 1; index < 10; index++ { // We skipped the first element here.
        if people[index].age > older.age { // Current's persons age vs olderest so far.
            older = people[index] // If people[index] is older, replace the value of older.
        }
    }
    return older
}

func main() {
    // Declare an example array variable of 10 person called 'array'.
    var array [10]person

    // Initialize some of the elements of the array, the others are by
    // default set to person{"", 0}
```

```
    array[1] = person{"Paul", 23};
    array[2] = person{"Jim", 24};
    array[3] = person{"Sam", 84};
    array[4] = person{"Rob", 54};
    array[8] = person{"Karl", 19};

    older := Older10(array) // Call the function by passing it our array.

    fmt.Println("The older of the group is: ", older.name)
}
```

Output:

The older of the group is: Sam

We could have declared and initialized the variable `array` of the `main()` function above in a single shot like this:

```
// Declare and initialize an array A of 10 person.
array := [10]person {
    person{"", 0},
    person{"Paul", 23},
    person{"Jim", 24},
    person{"Sam", 84},
    person{"Rob", 54},
    person{"", 0},
    person{"", 0},
    person{"", 0},
    person{"Karl", 10},
    person{"", 0}}
```

This means that to initialize an array, you put its elements between two braces, and you separate them with commas.

We can even omit the size of the array, and Go will count the number of elements given in the initialization for us. So we could have written the above code as:

```
// Declare and initialize an array of 10 persons, but let the compiler guess the size.
array := [...]person { // Substitute '...' instead of an integer size.
    person{"", 0},
    person{"Paul", 23},
    person{"Jim", 24},
    person{"Sam", 84},
    person{"Rob", 54},
    person{"", 0},
    person{"", 0},
    person{"", 0},
    person{"Karl", 10},
    person{"", 0}}
```

Specifically note the ellipsis `[...]`.

Note: **However**, the **size of an array is an important part of its definition**. They don't *grow* and they don't *shrink*. You can't, for example, have an array of 9 elements, and use it with a function that expects an array of 10 elements as an input. Simply because they are of *different* types. **Remember this.**

Some things to keep in mind:

- Arrays are values. Assigning one array to another, copies all of the elements of the right hand array to the left hand array. That is: if `A`, and `B` are arrays of the same type, and we write: `B = A`, then `B[0] == A[0]`, `B[1] == A[1]`... `B[n-1] == A[n-1]`.
- When passing an array to a function, it is copied. i.e. The function recieves a *copy* of that array, and **not** a reference (pointer) to it.
- Go comes with a built-in function `len` that returns variables sizes. In the previous example: `len(array) == 10`.

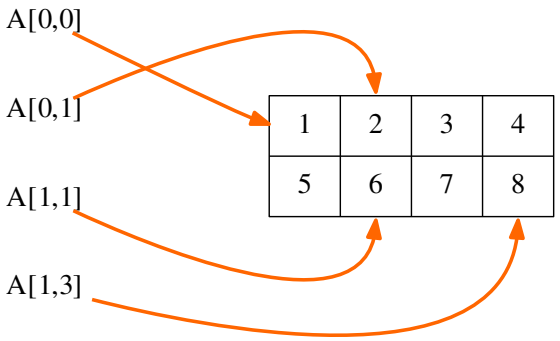
2.2.4 Multi-dimensional arrays

You can think to yourself: *"Hey! I want an array of arrays!"*. Yes, that's possible, and often used in practice.

We can declare a 2-dimensional array like this:

```
1 //declare and initialize an array of 2 arrays of 4 ints
2 double_array := [2][4]int {[4]int{1,2,3,4}, [4]int{5,6,7,8}}
```

This is an array of (2 arrays (of 4 `int`)). We can think of it as a matrix, or a table of two lines each made of 4 columns.



The previous declaration may be simplified, using the fact that the compiler can count arrays' elements for us, like this:

```
1 //simplify the previous declaration, with the '...' syntax
2 double_array := [2][4]int {[...]int{1,2,3,4}, [...]int{5,6,7,8}}
```

Guess what? Since Go is about cleaner code, we can simplify this code even further:

```
1 //über simpifikation!
2 double_array := [2][4]int {{1,2,3,4}, {5,6,7,8}}
```

Cool, eh? Now, we can combine multiple fields of different types to create a `struct` and we can have `arrays` of, as many as we want, of objects of the same type, and pass it as a *single block* to our functions. But this is just the begining! In the next chapter, we will see how to create and use some advanced composite data types with pointers.

2.3 Advanced composite types: Slices

We saw in the previous chapter how to combine some fields of different types in a container called a `struct` to create a new type that we can use in our functions and programs. We also saw how to gather **n** objects of the same type into a single array that we can pass as a single value to a function.

This is fine, but do you remember *the note we wrote about arrays*? Yes, that one about the fact that they don't grow or shrink, that an array of 10 elements of a given type is totally different of an array of, say, 9 elements of the same type.

This kind of *inflexibility* about arrays leads us to this chapter's subject.

2.3.1 What is a slice?

You'd love to have arrays without specifying sizes, right? Well, that is possible in a manner of speaking, we call 'dynamic arrays' in Go *slices*.

A slice is not actually a 'dynamic array' but rather a **reference** (think pointer) to a sub-array of a given array (anywhere from empty to all of it). You can declare a slice just like you declare an array, omitting the size.

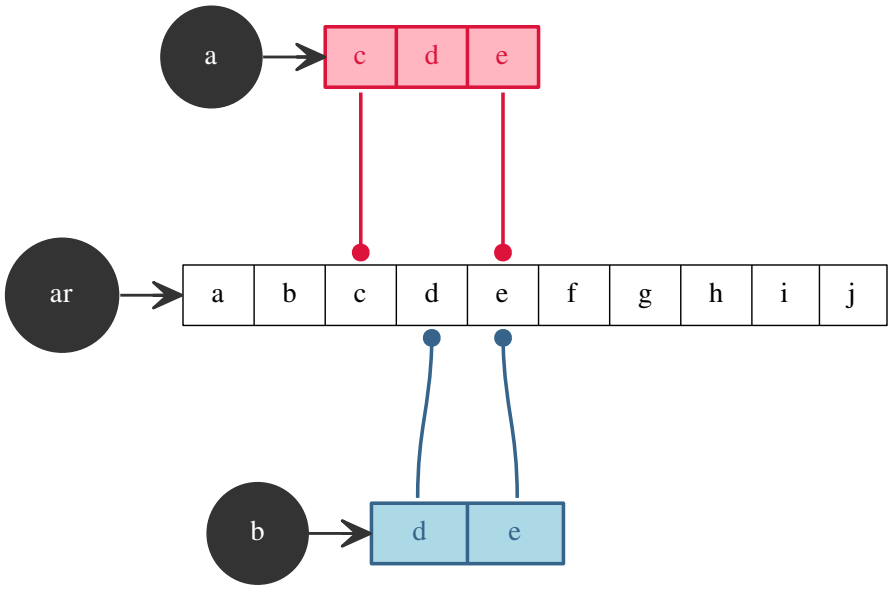
```
//declare a slice of ints
var array []int // Notice how we didn't specify a size.
```

As we see in the above snippet, we can declare a slice literal just like an array literal, except that we leave out the size number.

```
//declare a slice of ints
slice := []byte {'a', 'b', 'c', 'd'} // A slice of bytes (current size is 4).
```

We can create slices by *slicing* an array or even an existing slice. That is taking a portion (a 'slice') of it, using this syntax `array[i:j]`. Our created slice will contain elements of the array that start at index **i** and end before **j**. (`array[j]` not included in the slice)

```
// Declare an array of 10 bytes (ASCII characters). Remember: byte is uint8.
var array [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}

// Declare a and b as slice of bytes
var a_slice,b_slice []byte

// Make the slice "a_slice" refer to a "portion" of "array".
a_slice = array[2:5]
// a_slice is now a portion of array that contains: array[2], array[3] and array[4]

// Make the slice "b_slice" refer to another "portion" of "array".
b_slice = array[3:5]
// b_slice is now a portion of array that contains: array[3], array[4]
```



Slice shorthands

- When slicing, first index defaults to **0**, thus `ar[:n]` means the same as `ar[0:n]`.
- Second index defaults to len(array/slice), thus `ar[n:]` means the same as `ar[n:len(ar)]`.
- To create a slice from an entire array, we use `ar[:]` which means the same as `ar[0:len(ar)]` due to the defaults mentioned above.

Ok, let's go over some more examples to make this even clearer. Read slowly and pause to think about each line so that you fully grasp the concepts.

```
// Declare an array of 10 bytes (ASCII characters). Remember: byte is uint8
var array [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
//declare a and b as slice of bytes
var a_slice,b_slice []byte

// Examples of slicing
a_slice = array[4:8]
// means: a_slice contains elements of array from array[4] to array[7]: e,f,g,h
a_slice = array[6:7]
// means: a_slice contains ONE element of array and that is array[6]: g

// Examples of shorthands
a_slice = array[:3] // means: a_slice = array[0:3] thus: a contains: a,b,c
a_slice = array[5:] // means: a_slice = array[5:9] thus: a contains: f,g,h,i,j
a_slice = array[:] // means: a_slice = array[0:9] thus: a contains all array elements.

```

```
// Slice of a slice
a_slice = array[3:7] // means: a_slice contains elements form 3 to 6 of array: d,e,f,g
b_slice = a_slice[1:3] //means: b_slice contains elements a[1], a[2] i.e the chars e,f
b_slice = a_slice[:3] //means: b_slice contains elements a[0], a[1], a[2] i.e the chars: d,e,f
b_slice : a_slice[:] //mens: b_slice contains all elements of slice a: d,e,f,g
```

Does it make more sense now? Fine!

Is that all there is? What's so interesting about slices?

Lets show you more examples below for the yummy `sliced` fruity goodness.

2.3.2 Slices and functions

Say we need to find the biggest value in an array of 10 elements. Good, we know how to do it. Now, imagine that our program has the task of finding the biggest value in many arrays of different sizes! Aha! See? One function is not enough for that, because, remember, the types: `[n]int` and `[m]int` are **different**! They can not be used as an input of a single function.

Slices fix this quite easily! In fact, we need only write a function that accepts a slice of integers as an input parameter, and create slices of arrays.

Let's see the code.

```
package main
import "fmt"

// Return the biggest value in a slice of ints.
func Max(slice []int) int { // The input parameter is a slice of ints.
    max := slice[0] // The first element is the max for now.
    for index := 1; index < len(slice); index++ {
        if slice[index]>max { // We found a bigger value in our slice.
            max = slice[index]
        }
    }
    return max
}

func main() {
    // Declare three arrays of different sizes, to test the function Max.
    A1 := [10]int {1,2,3,4,5,6,7,8,9}
    A2 := [4]int {1,2,3,4}
    A3 := [1]int {1}

    // Declare a slice of ints.
    var slice []int

    slice = A1[:] // Take all A1 elements.
    fmt.Println("The biggest value of A1 is", Max(slice))
    slice = A2[:] // Take all A2 elements.
    fmt.Println("The biggest value of A2 is", Max(slice))
    slice = A3[:] // Take all A3 elements.
    fmt.Println("The biggest value of A3 is", Max(slice))
}
```

Output:

The biggest value of A1 is 9
The biggest value of A2 is 4
The biggest value of A3 is 1

You see? Using a slice as the input parameter of our function made it *reusable* and we didn't had to rewrite the same function for arrays of different sizes. So remember this: whenever you think of writing a function that takes an array as its input, think again, and use a *slice* instead.

Let's see other advantages of slices.

2.3.3 Slices are references

We stated earlier that slices are *references* to some underlying array (or another slice). What exactly does that mean? It means that slicing an array or another slice doesn't simply copy some of the array's elements into the new slice, it instead make the new slice *point* to the element of this sliced array similar to the way pointers operate.

In other words: changing an element's value in a slice will actually change the value of the element in the underlying array to which the slice points. This changed value will be visible in all slices and slices of slices containing the array element.

Let's see an example:

```
package main
import "fmt"

func PrintByteSlice(name string, slice []byte){
    fmt.Printf("%s is : [", name)
    for index :=0; index < len(slice)-1; index ++{
        fmt.Printf("%q,", slice[i])
    }
    fmt.Printf("%q]\n", slice[len(slice)-1])
}

func main(){
    // Declare an array of 10 bytes.
    A := [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}

    //declare some slices.
    slice1 := A[3:7] // Slice1 == {'d', 'e', 'f', 'g'}
    slice2 := A[5:] // Slice2 == {'f', 'g', 'h', 'i', 'j'}
    slice3 := slice1[:2] // Slice3 == {'d', 'e'}

    // Let's print the current content of A and the slices.
    fmt.Println("\nFirst content of A and the slices")
    PrintByteSlice("A", A[:])
    PrintByteSlice("slice1", slice1)
    PrintByteSlice("slice2", slice2)
    PrintByteSlice("slice3", slice3)

    // Let's change the 'e' in A to 'E'.
    A[4] = 'E'
    fmt.Println("\nContent of A and the slices, after changing 'e' to 'E' in array A")
    PrintByteSlice("A", A[:])
    PrintByteSlice("slice1", slice1)
    PrintByteSlice("slice2", slice2)
    PrintByteSlice("slice3", slice3)
```

```
35
36      // Let's change the 'g' in slice2 to 'G'.
37      slice2[1] = 'G' // Remember that 1 is actually the 2nd element in slice2!
38      fmt.Println("\nContent of A and the slices, after changing 'g' to 'G' in slice2")
39      PrintByteSlice("A", A[:])
40      PrintByteSlice("slice1", slice1)
41      PrintByteSlice("slice2", slice2)
42      PrintByteSlice("slice3", slice3)
43  }
```

Output:

First content of A and the slices
A is : ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
slice1 is : ['d', 'e', 'f', 'g']
slice2 is : ['f', 'g', 'h', 'i', 'j']
slice3 is : ['d', 'e']

Content of A and the slices, after changing 'e' to 'E' in array A
A is : ['a', 'b', 'c', 'd', 'E', 'f', 'g', 'h', 'i', 'j']
slice1 is : ['d', 'E', 'f', 'g']
slice2 is : ['f', 'g', 'h', 'i', 'j']
slice3 is : ['d', 'E']

Content of A and the slices, after changing 'g' to 'G' in slice2
A is : ['a', 'b', 'c', 'd', 'E', 'f', 'G', 'h', 'i', 'j']
slice1 is : ['d', 'E', 'f', 'G']
slice2 is : ['f', 'G', 'h', 'i', 'j']
slice3 is : ['d', 'E']

Let's talk about the `PrintByteSlice` function a little bit. If you analyze its code, you'll see that it loops until before `len(slice)-1` to print the element `slice[i]` followed by a comma. At the end of the loop, it prints `slice[len(slice)-1]` (the last item) with a closing bracket instead of a comma.

`PrintByteSlice` works with slices of bytes, of any size, and can be used to print arrays content also, by using a slice that contains all of its elements as an input parameter. This proves the utility of the advice before: always think of *slices* when you're about to write functions *for arrays*.

Until now, we've learned that slices are truly useful as input parameters for functions which expect *blocks* of a variable size of consecutive elements of the same type. We have learned also that slices are references to portions of an array or another slice.

That is not all. Here's an awesome feature of slices.

2.3.4 Slices are resizable

One of our concerns with arrays is that they're inflexible. Once you declare an array, its size can't change. It won't shrink nor grow. How unfortunate!

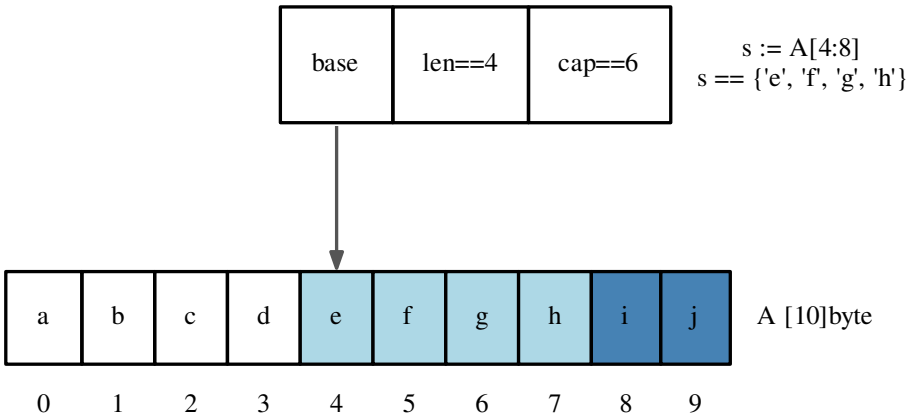
Slices can grow or shrink.

How so? Conceptually, a *slice* is a `struct` of three fields:

- A *pointer* to the first element of the array *where the slice begins*.
- A *length* which is an **int** representing the total number of *elements in the slice*.
- A *capacity* which is an **int** representing the number of *available elements*.

```
type Slice struct {
    base *elementType //the pointer
    len int //length
    cap int //capacity
}
```

Schematically, since you *love* my diagrams.




```
// Declare an array of 10 bytes.
array := [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
slice := A[4:8]
```

The picture above is a representation of a slice that starts from the 5th element of the array. It contains exactly 4 elements (its length) and can contain up to 6 elements on the same underlying array (its capacity) starting from the first element of the slice.

Philosophy

Simply said: Slicing does not make a copy of the elements, it just creates a new structure holding a different pointer, length, and capacity.

To make things even easier, Go comes with a built-in function called make, which has the signature: `func make([]T, len, cap) []T`.

Where `T` stands for the element type of the slice to be created. The make function takes a *type*, a *length*, and an *optional capacity*. When called, make allocates an array and returns a slice that refers to that array.

The optional capacity parameter, when omitted, defaults to the specified length. These parameters can be inspected for a given slice, using the built-in functions: - `len(slice)` which returns a slice's length, and - `cap(slice)` which returns a slice's capacity.

Example:

```
var slice1, slice2 []int
slice1 = make([]int, 4, 4) // slice1 is []int {0, 0, 0, 0}
slice2 = make([]int``,`` 4) // slice2 is []int {0, 0, 0, 0}
// Note: cap == len == 4 for slice2
// cap(slice2) == len(slice2) == 4
```

The zero value of a slice is `nil`. The `len` and `cap` functions will both return **0** for a `nil` slice.

```
package main

import "fmt"

func main() {
    var slice []int
    fmt.Println("Before calling make")
    if slice==nil {fmt.Println("slice == nil")}
    fmt.Println("len(slice) == ", len(slice))
    fmt.Println("cap(slice) == ", cap(slice))

    // Let's allocate the underlying array:
    fmt.Println("After calling make")
    slice = make([]int, 4)
    fmt.Println("slice == ", slice)
    fmt.Println("len(slice) == ", len(slice))
    fmt.Println("cap(slice) == ", cap(slice))

    // Let's change things:
    fmt.Println("Let's change some of its elements: slice[1], slice[3] = 2, 3")
    slice[1], slice[3] = 2, 3
    fmt.Println("slice == ", slice)
}
```

Output:

Before calling make
slice == nil
len(slice) == 0
cap(slice) == 0
After calling make
slice == [0 0 0 0]
len(slice) == 4
cap(slice) == 4
Let's change some of its elements: slice[1], slice[3] = 2, 3
slice == [0 2 0 3]

We may find ourselves wondering "Why use `make` instead of `new` that we discussed when we studied *pointers*?"

Let's discuss the reason why we use `make` and not `new` to create a new slice:

Recall that `new(T)` allocates memory to store a variable of type `T` and returns a pointer to it. Thus `new(T)` returns `*T`, right?

So for example if we had used `new([]int)` to create a slice of integers, `new` will allocate the internal structure discussed above and that will consist of a pointer, and two integers (length and capacity). `new` will then return us a pointer of the type `*[]int` – which... let's face it... is not what we *need*!

What we do *need* is a function which:

- Returns the actual structure not a pointer to it. i.e. returns `[]int` not `*[]int`.
- Allocates the underlying array that will hold the slice's elements.

Do you see the difference? `make` doesn't merely allocate enough space to *host* the structure of a given slice (the pointer, the length and the capacity) only, it allocates both the structure *and* the underlying array.

Back to our resizability of slices. As you may guess, shrinking a slice is really easy! All we have to do is reslice it and assign the new slice to it.

Now, let's see how – using only what we have learned until now – we grow a slice. A very simple, straight forward way of growing a slice might be:

1. make a new slice
2. copy the elements of our slice to the one we created in 1
3. make our slice refer to the slice created in 1

```
package main
import "fmt"

func PrintIntSlice(name string, slice []int){
    fmt.Printf("%s == [", name)
    for index := 0; index < len(slice)-1; index++ {
        fmt.Printf("%d,", slice[index])
    }
    fmt.Printf("%d]\n", slice[len(slice)-1])
}


func GrowIntSlice(slice []int, add int) []int {
    new_capacity := cap(slice)+add
    new_slice := make([]int, len(slice), new_capacity)
    for index := 0; index < len(slice); index++ {
        new_slice[index] = slice[index]
    }
    return new_slice
}

func main(){
    slice := []int{0, 1, 2, 3}

    fmt.Println("Before calling GrowIntSlice")
    PrintIntSlice("slice", slice)
    fmt.Println("len(slice) == ", len(slice))
    fmt.Println("cap(slice) == ", cap(slice))

    // Let's call GrowIntSlice
    slice = GrowIntSlice(slice, 3) //add 3 elements to the slice

    fmt.Println("After calling GrowIntSlice")
    PrintIntSlice("slice", slice)
```

```
35      fmt.Println("len(slice) == ", len(slice))
36      fmt.Println("cap(slice) == ", cap(slice))
37
38      // Let's two elements to the slice
39      // So we reslice the slice to add 2 to its original length
40      slice = slice[:len(slice)+2] // We can do this because cap(slice) == 7
41      slice[4], slice[5] = 4, 5
42      fmt.Println("After adding new elements: slice[4], slice[5] = 4, 5")
43      PrintIntSlice("slice", slice)
44      fmt.Println("len(slice) == ", len(slice))
45      fmt.Println("cap(slice) == ", cap(slice))
46  }
```

Output:

Before calling GrowIntSlice
slice == [0,1,2,3]
len(slice) == 4
cap(slice) == 4
After calling GrowIntSlice
slice == [0,1,2,3]
len(slice) == 4
cap(slice) == 7
After adding new elements: slice[4], slice[5] = 4, 5
slice == [0,1,2,3,4,5]
len(slice) == 6
cap(slice) == 7

Let's discuss this program a little bit.

First, the `PrintIntSlice` function is similar to the `PrintByteSlice` function we saw earlier. Except that we improved it to support `nil` slices.

Then the most important one is `GrowIntSlice` which takes two input parameters: a `[]int` slice and an `int add` that represents the amount of elements to add to the capacity.

GrowIntSlice first `makes` a new slice with a `new_capacity` that is equal the original slice's capacity plus the `add` parameter.

Then it copies elements from `slice` to `new_slice` in the highlighted loop in lines 16, 17 and 18.

And finally, it returns the `new_slice` as an output.

The approach you may be used to or that first comes to mind is often implemented as a built-in Go function provided for you already. This is most definitely one such example. There is a built-in Go function called `copy` which takes two arguments: source and destination, and copies elements from the source to the destination and returns the number of elements copied.

The signature of `copy` is: `func copy(dst, src []T) int`

So we can replace the lines 16, 17, 18 with this very simple one:

```
1  // Instead of lines 16, 17, 18 of the previous source:
2  copy(new_slice, slice) // Copy elements from slice to new_slice
```

You know what? Let's write another example and use the `copy` function in it.

```
package main
import "fmt"

func PrintByteSlice(name string, slice []byte){
    fmt.Printf("%s is : [", name)
    for index := 0; index < len(slice)-1; index++ {
        fmt.Printf("%q,", slice[index])
    }
    fmt.Printf("%q]\n", slice[len(slice)-1])
}

func Append(slice, data []byte) []byte {
    length := len(slice)
    if length + len(data) > cap(slice) {  // Reallocate
        // Allocate enough space to hold both slice and data
        newSlice := make([]byte, 1+len(data))
        // The copy function is predeclared and works for any slice type.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:length+len(data)]

    copy(slice[length:length+len(data)], data)
    return slice
}

func main() {
    hello := []byte {'H', 'e', 'l', 'l', 'o'}
    world := []byte {' ', 'W', 'o', 'r', 'l', 'd'}
    fmt.Println("Before calling Append")
    PrintByteSlice("hello", hello)
    PrintByteSlice("world", world)

    fmt.Println("After calling Append")
    Append(hello, world)
    PrintByteSlice("hello", hello)
    PrintByteSlice("world", world)
}
```

Output:

Before calling Append
hello is : ['H','e','l','l','o']
world is : ['W','o','r','l','d']
After calling Append
hello is : ['H','e','l','l','o','W','o','r','l','d']
world is : ['W','o','r','l','d']

Appending to a slice is a very common operation and as such Go has a built-in function called `append` that we will see in details very soon.

Phew, that was a long chapter, eh? We'll stop here for now, go out, have a nice day, and see you tomorrow for another interesting composite type. If you are completely enthralled thus far by this exquisite book then do yourself a favor

before continuing and go get a tea or coffee and take a nice 10 minute break.

2.4 Advanced composite types: Maps

This chapter will be lighter than the previous one, so don't worry, and smile for the perspective of adding a new tool to your box.

Arrays and slices are wonderful tools for collecting elements of the same type in a *sequential* fashion. We simply say: the first element, the second element, ... the **n** th element of the array or the slice. We can access and modifiy elements using their numerical integer indices in the `array` or the *slice*.

This is nice and quite useful. Now suppose that we'd like to access and modify elements given a name of some type (a non-integer name or 'index'). For example: Say we wish to access the definition of the word "Hello" in a dictionary or to find out the capital of "Japan".

These category of problems call for a new kind of type. A type where we can specify a *key* from which a *value* will be stored and retrieved. It's not about the nth element of a sequence, it's about an *association* of things (keys) with other things (values).

This kind of types is called a *dict* in Python and *map* in Go and a *hash* in Ruby.

2.4.1 How to declare a map?

The syntax of a map type is: `map[keyType] valueType`.

For example:

```
// A map that associates strings to int
// eg. "one" --> 1, "two" --> 2...
var numbers map[string] int //declare a map of strings to ints
```

You can access and assign a value to an *entry* of this map using the square bracket syntax as in arrays or slices, but instead of an `int` index you use a key of type `keyType`.

As with slices, since maps are reference types, we can make them using the `make` function: `make(map[string]float32)`.

When used with maps, `make` takes an optional capacity parameter.

```
// A map that associates strings to int
// eg. "one" --> 1, "two" --> 2...
var numbers map[string] int //declare a map of strings to ints
numbers = make(map[string]int)
numbers["one"] = 1
numbers["ten"] = 10
numbers["trois"] = 3 //trois is "three" in french. I know that you know.
//...
fmt.Println("Trois is the french word for the number: ", numbers["trois"])
// Trois is the french word for the number: 3. Also a good time.
```

We now have the idea: it's like a table with two columns: in the left column we have the key, and on the right column we have its *associated* value.

Key	Value
"one"	1
"ten"	10
"trois"	3

numbers

Some things to notice:

- There is no defined notion of *order*. We do not access a value by an index, but rather by a *key* instead.
- The size of a map is not fixed like in arrays. In fact, just like a *slice*, a map is a *reference* type.
- Doing for example `numbers["coffees_I_had"] = 7` will actually add an entry to this table, and the size of the map will be incremented by 1.
- As in slices and arrays, the built-in function `len` will return the number of keys in a map (thus the number of entries).
- Values can be changed, of course. `numbers["coffees_I_had"] = 12` will change the `int` value associated with the string "coffes_I_had".

Literal values of maps can be expressed using a list of colon-separated `key:value` pairs. Let's see an example of this:

```
// A map representing the rating given to some programming languages.
rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }
// This is equivalent to writing more verbosely
var rating = map[string]float32
rating = make(map[string]float)
rating["C"] = 5
rating["Go"] = 4.5
rating["Python"] = 4.5
rating["C++"] = 2 //Linus would put 1 at most. Go ask him
```

Maps are references

If you assign a map `m` to another map `m1`, they will both refer to the same underlying structure that holds the key/value pairs. Thus, changing the value associated with a given key in `m1` will also change the value of that key in `m` as they

both reference the same underlying data:

```
//let'ss ay a translation dictionary
m = make(map[string]string)
m["Hello"] = "Bonjour"
m1 = m
m1["Hello"] = "Salut"
// Now: m["Hello"] == "Salut"
```

Checking existence of a key

Question What would the expression `rating["C#"]` return as a value, in our previous example?

Good question! The answer is simple: the expression will return the zero value of the value type.

The value type in our example is `float32`, so it will return '0.00'.

```
//A map representing the rating given to some programming languages.
rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }
csharp_rating := rating["C#"]
//csharp_rating == 0.00
```

But then, if the value associated with an inexistent key is the zero of the type value, how can we be sure that C#'s rating is actually 0.00? In other words: is C#'s rating actually 0.00 so we can say that as a language it *stinks* or was it that it was not rated at all?

Here comes the "comma-ok" form of accessing a key's associated value in a map. It has this syntax: `value, present = m[key]`. Where `present` is a boolean that indicates whether the key is present in the map.

```
//A map representing the rating given to some programming languages.
rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }
csharp_rating, ok := rating["C#"]
//would print: We have no rating associated with C# in the map
if ok {
    fmt.Println("C# is in the map and its rating is ", csharp_rating)
} else {
    fmt.Println("We have no rating associated with C# in the map")
}
```

We often use `ok` as a variable name for boolean presence, hence the name "comma-ok" form. But, hey! You're free to use any name as long as it's a `bool` type.

Deleting an entry

To delete an entry from the map, think of an *inversed* "comma-ok" form. In fact, you just have to assign any given value followed by comma `false`.

```
// A map representing the rating given to some programming languages.
rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }
map["C++"] = 1, false // We delete the entry with key "C++"
cpp_rating, ok := rating["C++"]
// Would print: We have no rating associated with C++ in the map
if ok {
    fmt.Println("C++ is in the map and its rating is ", cpp_rating)
} else {
    fmt.Println("We have no rating associated with C++ in the map")
}
```

If in line 2, we had `map["C++"] = 1, true` the output of the if-else statement would be: *C++ is in the map and its rating is 1*. i.e. the entry associated with key "C++" would be kept in the map, and its value changed to 1.

Cool! We now have a sexy new type which allows us to easily add key/value pairs, check if a given key is present, and delete any given key. Simply.

The question now is: "How do I retrieve all the elements in my map?" More specifically, "How do I print a list of all the languages in the `rating` map with their respective ratings?"

2.4.2 The range clause

For maps (and arrays, and slices, and other stuff which we'll see later), Go comes with a facinating alteration of the syntax for the "for statement".

This syntax is as follow:

```
for key, value := range m {
    // In each iteration of this loop, the variables key and value are set
    // to the current key/value in the map
    ...
}
```

Let's see a complete example to understand this better.

```
package main
import "fmt"

func main(){
    // Declare a map literal
    ratings := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }

    // Iterate over the ratings map
    for key, value := range ratings {
        fmt.Printf("%s language is rated at %g\n", key, value)
    }
}
```

Output:

C++ language is rated at 2
C language is rated at 5
Go language is rated at 4.5
Python language is rated at 4.5

If we don't need the value in our for statement, we can omit it like this:

```
package main
import "fmt"

func main(){
    // Declare a map literal.
    ratings := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }

    fmt.Print("We rated these languages: ")

    // Iterate over the ratings map, and print the languages names.
```

```
11      for key := range ratings {
12          fmt.Print(key, ",")
13      }
14  }
```

Output:

We rated these languages: C++,C,Go,Python,

Exercise: Modify the program above to replace the last comma in the list by a period. That is, output: "We rated these languages: C++,C,Go,Python."

This "for statement" form is also available for arrays and slices where instead of a key we have an index.

Let's rewrite a previous example using this new tool:

```
1   package main
2   import "fmt"
3
4   // Return the biggest value in a slice of ints.
5   func Max(slice []int) int { // The input parameter is a slice of ints.
6       max := slice[0] //the first element is the max for now.
7       for index, value := range slice { // Notice how we iterate!
8           if value > max { // We found a bigger value in our slice.
9               max = value
10          }
11      }
12      return max
13  }
14
15  func main() {
16      // Declare three arrays of different sizes, to test the function Max.
17      A1 := [10]int {1,2,3,4,5,6,7,8,9}
18      A2 := [4]int {1,2,3,4}
19      A3 := [1]int {1}
20
21      //declare a slice of ints
22      var slice []int
23
24      slice = A1[:] // Take all A1 elements.
25      fmt.Println("The biggest value of A1 is", Max(slice))
26      slice = A2[:] // Ttake all A2 elements.
27      fmt.Println("The biggest value of A2 is", Max(slice))
28      slice = A3[:] // Ttake all A3 elements.
29      fmt.Println("The biggest value of A3 is", Max(slice))
30  }
```

Output:

The biggest value of A1 is 9
The biggest value of A2 is 4
The biggest value of A3 is 1

Look carefully at line 6. We used a `range` over a slice of ints. `i` is an index and it goes from 0 to `len(s)-1` and

`value` is an int that goes from `s[0]` to `s[len(s)-1]`.

Notice also how we didn't use the index `i` in this loop. We didn't need it.

2.4.3 The blank identifier

Whenever a function returns a value you don't care about, or a range returns an index that you don't care about, you can use the *blank identifier* _ (underscore) This predeclared identifier can be assigned any value of any type, and it will be discarded.

We could have written the `Max(s []int)int` function like this:

```
// Return the biggest value in a slice of ints.
func Max(slice []int) int { // The input parameter is a slice of ints.
    max := slice[0]
    for _, value := range slice { // Notice how we use _ to "ignore" the index.
        if value > max {
            max = value
        }
    }
    return max
}
```

Also, say, we have a function that returns two or more values of which some are unimportant for us. We can "ignore" these output results using the *blank identifier*.

```
// A function that returns a bool that is set to true of Sqrt is possible
// and false when not. And the actual square root of a float64
func MySqrt(floater float64) (squareroot float64, ok bool) {
    if floater > 0 {
        squareroot, ok = math.Sqrt(f), true
    } else {
        squareroot, ok = 0, false
    }
    return squareroot, ok
}
//...
r _ = MySqrt(v) //retrieve the square root of v, and ignore its faisability
```

That's it for this chapter. We learned about *maps* ; how to make them, how to add, change, and delete key/value pairs from them. How to check if a given key exists in a given map. And we also learned about the *blank identifier* '_' and the `range` clause.

Yes, the `range` clause, especially, is a control flow construct that should belong in the chapter about control flow, but we didn't yet know about arrays, slices or maps, at that time. This is why we deferred it until this chapter.

The next chapter will be about things that we didn't mention about functions in the chapter about functions for the same reason: lack of prior exposure, or simply because I wanted the chapter to be light so we can make progress with advanced data structures.

Anyways, you'll see, it will be fun! See you in the next chapter! :)

CHAPTER 3

And be smart and awesome

We will learn smarter and nicer things about functions, and will have fun writing code to represent and solve real world problems (well... real as in objects. — You know what I mean)

3.1 Getting funky with functions

In the previous chapters, we went from how to declare variables, pointers, how to write basic control structures, how to combine these control structures to write functions. Then we went back to data, and how to combine basic data types to create composite data types.

And now, armed with this knowledge, we're ready to work on advanced aspects of functions.

I decided to divide this task in two chapters, not because the things we're about to study are hard, but mainly because it's easier, and more fun to work on a few topics at a time.

3.1.1 Variadic functions

Do you remember when I made you *cringe* at the idea of writing a function `Older100` that will not accept less than 100 `persons`? `structs` as its input?

Well I lied a little bit, but it was a good lie. I duct-taped you to a chair with that blindfold on with the best of intentions, I swear! It was an excellent reason to learn about arrays, and later about slices! Relax... it's for your own safety! ;)

The truth is: you can have functions that accept a *variable* number of input parameters of the *same* type. They're called *variadic functions*.

These functions can be declared like this:

```
func function_name(args ...inputType) (output1 OutputType1 [, output2
OutputType2 [, ...]])
```

There's no difference from the way we learned to declare functions, except the three dots (called an ellipses) thing: `...inputType` which says to the function that it can recieve a variable number of parameters all of them of type `inputType`.

How does this work? Easy: In fact, the `args` identifier you see in the input parameter is actually a slice which contains all of the parameters passed in (which should all be of type `inputType`).

A slice! Yay! So we can, in the function's body, iterate using `range` to get all the parameters! Yes, that's the way you do it. You play the guitar on the mtv... er, sorry. I got a case of *Mark Knopfler* fever.

Let's see an example:

```go
package main
import "fmt"

// Our struct representing a person
type person struct {
    name string
    age int
}

// Return true, and the older person in a group of persons
// Or false, and nil if the group is empty.
func Older(people ...person) (bool, person) { // Variadic function.
    if(len(people) == 0){return false, person{}} // The group is empty.
    older := people[0] // The first one is the older FOR NOW.
    // Loop through the slice people.
    for _, value := range people { // We don't need the index.
        // Compare the current person's age with the oldest one so far
        if value.age > older.age {
            older = value //if value is older, replace older
        }
    }
    return true, older
}

func main(){

    // Two variables to be used by our program.
    var(
        ok bool
        older person
    )

    // Declare some persons.
    paul := person{"Paul", 23}
    jim := person{"Jim", 24}
    sam := person{"Sam", 84}
    rob := person{"Rob", 54}
    karl := person{"Karl", 19}

    // Who is older? Paul or Jim?
    _, older = Older(paul, jim) //notice how we used the blank identifier
    fmt.Println("The older of Paul and Jim is: ", older.name)
    // Who is older? Paul, Jim or Sam?
    _, older = Older(paul, jim, sam)
    fmt.Println("The older of Paul, Jim and Sam is: ", older.name)
    // Who is older? Paul, Jim , Sam or Rob?
    _, older = Older(paul, jim, sam, rob)
    fmt.Println("The older of Paul, Jim, Sam and Rob is: ", older.name)
    // Who is older in a group containing only Karl?
    _, older = Older(karl)
    fmt.Println("When Karl is alone in a group, the older is: ", older.name)
    // Is there an older person in an empty group?
    ok, older = Older() //this time we use the boolean variable ok
    if !ok {
        fmt.Println("In an empty group there is no older person")
    }
}
```

Output:

```
The older of Paul and Jim is: Jim
The older of Paul, Jim and Sam is: Sam
The older of Paul, Jim, Sam and Rob is: Sam
When Karl is alone in a group, the older is: Karl
In an empty group there is no older person
```

Look how we called the function `Older` with 2, 3, 4, 1 and even no input parameters at all. We could have called it with 100 `persons` if we wanted to.

Variadic functions are easy to write and can be handy in a lot of situations. By the way, I promised to tell you about the built-in function called `append` that makes appending to slices easier! Now, I can tell you more, because, guess what? The built-in `append` function is a variadic function!

The built-in append function

The append function has the following signature:

`func append(slice []T, elements...T) []T.`

It takes a slice of type `[]T`, and as many other `elements` of the same type `T` and returns a new slice of type `[]T` which includes the given elements.

Example:

```
package main
import "fmt"

func main(){
    slice := []int {1, 2, 3}
    fmt.Println("At first: ")
    fmt.Println("slice = ", slice)
    fmt.Println("len(slice) = ", len(slice))
    fmt.Println("Let's append 4 to it")
    slice = append(slice, 4)
    fmt.Println("slice = ", slice)
    fmt.Println("len(slice) = ", len(slice))
    fmt.Println("Let's append 5 and 6 to it")
    slice = append(slice, 5, 6)
    fmt.Println("slice = ", slice)
    fmt.Println("len(slice) = ", len(slice))
    fmt.Println("Let's append 7, 8, and 9 to it")
    slice = append(slice, 7, 8, 9)
    fmt.Println("slice = ", slice)
    fmt.Println("len(slice) = ", len(slice))
}
```

Output:

```
At first:
slice = [1 2 3]
len(slice) = 3
```

Let's append 4 to it
slice = [1 2 3 4]
len(slice) = 4
Let's append 5 and 6 to it
slice = [1 2 3 4 5 6]
len(slice) = 6
Let's append 7, 8, and 9 to it
slice = [1 2 3 4 5 6 7 8 9]
len(slice) = 9

You can even give a variadic function a slice of type `[]T` instead of a list of comma-separated parameters of type `T`.

Example:

```
// Two slices of ints
a_slice := []int {1, 2, 3}
b_slice := []int {10, 11, 12}
a_slice = append(a_slice, b_slice...) // Note the syntax: the slice followed by three dots
// a_slice == {1, 2, 3, 10, 11, 12}
```

Again, just in case you glossed over the comment, notice the ellipses '`...`' immediately following the second slice argument to the variadic function.

Using the append function to delete an element

Suppose that we have a slice `s` and that we would like to delete an element from it. There are 3 cases to consider:

- The element we'd like to delete is the first one of the slice.
- The element we'd like to delete is the last one of the slice
- The element we'd like to delete is the one at index `i` of the slice (where `i` is between the first and the last ones)

Let's write a function that deletes the element at a given index `i` in a given slice of `ints` and see how the `append` function can help.

```
package main
import "fmt"

func delete(i int, slice []int) []int{
    switch i {
        case 0: slice = slice[1:]
        case len(slice)-1: slice = slice[:len(slice)-1]
        default: slice = append(slice[:i], slice[i+1:]...)
    }
    return slice
}

func main(){
    slice := []int {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    fmt.Println("In the beginning...")
    fmt.Println("slice = ", slice)
    fmt.Println("Let's delete the first element")
    slice = delete(0, slice)
    fmt.Println("slice = ", slice)
    fmt.Println("Let's delete the last element")
```

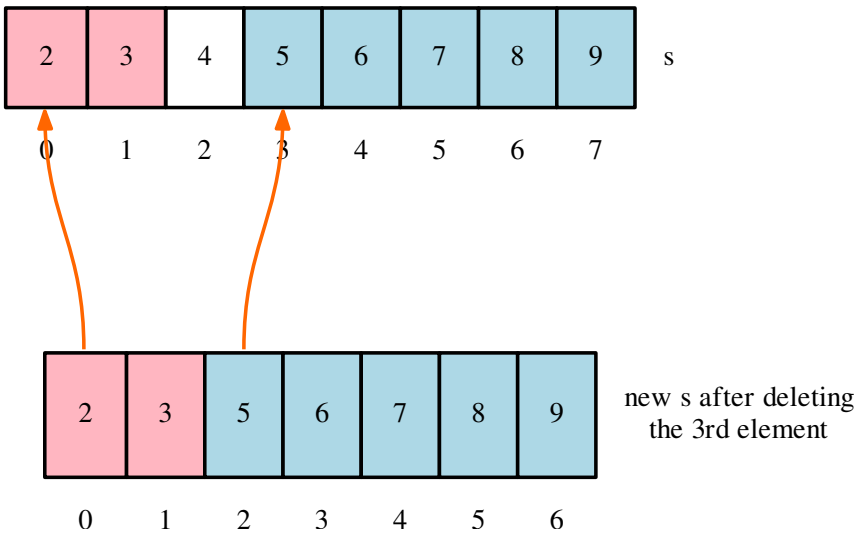
```
21      slice = delete(len(slice)-1, slice)
22      fmt.Println("slice = ", slice)
23      fmt.Println("Let's delete the 3rd element")
24      slice = delete(2, slice)
25      fmt.Println("slice = ", slice)
26  }
```

Output:

In the beginning...
slice = [1 2 3 4 5 6 7 8 9 10]
Let's delete the first element
slice = [2 3 4 5 6 7 8 9 10]
Let's delete the last element
slice = [2 3 4 5 6 7 8 9]
Let's delete the 3rd element
slice = [2 3 5 6 7 8 9]

The lines 6 and 7 of our program are fairly easy to understand. Aren't they? We re-slice our slice omitting the first and the last elements respectively.

Now, the case where the element is not the first nor the last one. We assign to our slice the result of `append` of the slice starting from the first up to, but not including, the `i` th element and the slice that starts from the element after the `i` th (i.e. i+1) up to the last one. That is, we made our slice contain elements from the right sub-slice and the left one to the element that we want to delete.



Simple, isn't it? Now, we actually complicated the `delete` function in vain. We could have written it like this:

```
//simpler delete
func delete(i int, slice []int) []int {
    slice = append(slice[:i], slice[i+1:]...)
    return slice
}
```

Yes, it will work. Go and try it. Can you figure out why? (**Hint**: empty. **Hint Hint**: empty.)

3.1.2 Recursive functions

Some algorithmic problems can be thought of in a beautiful way using recursion!

A function is said to be *recursive* when it calls itself within it's own body. For example: the Max value in a slice is the maximum of the Max value of two sub-slices of the same slice, one starting from 0 to the middle and the other starting from the middle to the end of the slice. To find out the Max in each sub-slice we use the same function, since a sub-slice is itself a slice!

Enough talk already, let's see this wonderous beast in action!

```
package main
import "fmt"

//Recursive Max
func Max(slice []int) int {
    if len(slice) == 1 {
        return slice[0] //there's only one element in the slice, return it!
```

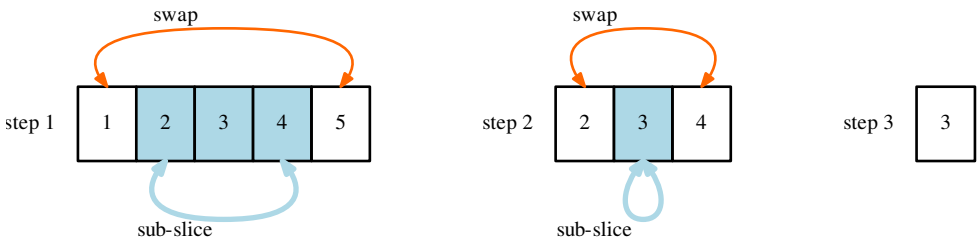
```
    }

    middle := len(slice)/2 //the middle index of the slice
    //find out the Max of each sub-slice
    m1 := Max(slice[:middle])
    m2 := Max(slice[middle:])
    //compare the Max of two sub-slices and return the bigger one.
    if m1 > m2 {
        return m1
    }
    return m2
}

func main(){
    s := []int {1, 2, 3, 4, 6, 8}
    fmt.Println("Max(s) = ", Max(s))
}
```

Output:

Max(s) = 8

Another example: How to invert the data in a given slice of ints? For example, given `s := []int {1, 2, 3, 4, 5}`, we want our program to modify `s` to be `{5, 4, 3, 2, 1}`.

A recursive strategy to do this involves first swapping the first and the last elements of `s` and then inverting the slice between these elements, i.e. the one from the 2nd element to the one just before the last one.



Let's write it:

```
package main
import "fmt"

func Invert(slice []int){
    length := len(slice)
    if length > 1 { // Only a slice of 2 or more elements can be inverted
        slice[0], slice[length-1] = slice[length-1], slice[0] // Swap first and last ones
        Invert(slice[1:length-1]) // Invert the slice in between
    }
}

```

```
12 func main(){
13     slice := []int {1, 2, 3, 4, 5}
14     fmt.Println("slice = ", slice)
15     Invert(slice)
16     fmt.Println("Invert(slice) = ", slice)
17 }
```

Output:

slice = [1 2 3 4 5]
Invert(slice) = [5 4 2 3 1]

Exercise Recursive absurdity! The SUM of the ints in a given slice is equal to the first element *plus* the SUM of the subslice starting from the second element. Go and write this program.

And these are but a few examples, many problems can be solved using recursion. You know, just like walking: to walk, you put one foot in front of the other and you walk :)

3.1.3 The defer statement

Sometimes you'd like to do *something* just before a `return` statement of a function, like for example closing an opened file. Now, suppose that our function has many spots where we call `return`, like for example in a `if`/`else` chain, or in a `switch` statement. In these cases we'd do that *something* as many times as there is a `return`.

The `defer` statement fixes this annoying irreverant mind-numbing repetition.

The `defer` statement schedules a function call (the deferred function) to be run immediately before the function executing the `defer` returns.

Let's see an example to make the idea clearer:

The os package comes with functions to open, close, create... files, Go figure!. Being a good lil programmer means that we have to close every file we open in our program after we've done our nefarious business with it's contents.

Now, imagine a function that opens a file, processes the data in it, but has many return spots in its body. Without `defer` you'll have to *manually* close any already opened file, just before the `return`.

Look at this simple program from Effective Go.

```
1  package main
2
3  import (
4      "os"
5      "fmt"
6  )
7
8  // Contents returns the file's contents as a string.
9  func Contents(filename string) (string, os.Error) {
10     f, err := os.Open(filename)
11     if err != nil {
12         return "", err
13     }
14     defer f.Close()  // f.Close will run when we're finished.
15
16     var result []byte
17     buf := make([]byte, 100)
```

```
18      for {
19          n, err := f.Read(buf[0:])
20          result = append(result, buf[0:n]...) // append is discussed later.
21          if err != nil {
22              if err == os.EOF {
23                  break
24              }
25              return "", err  // f will be closed if we return here.
26          }
27      }
28      return string(result), nil // f will be closed if we return here.
29  }
30
31
32  func main() {
33      contents, _ := Contents("/etc/hosts")
34      fmt.Println(contents)
35  }
```

Our function `Contents` needs a file name as its input, and it ouputs this file's content, and an eventual error (for example in case the file was not found).

On line 10, we open the file. If the returned error from `os.Open` is not `nil` then we were unable to open the file. So we return an empty string and the same error that `os.Open` returned.

Then on line 14, we `defer` the call to `f.Close()` so that the file will be *automatically* closed when the function emits a return (2 spots in our function).

Then we declare a `result` slice that will contain all the bytes read from the file. And we make a buffer `buf` of 100 bytes that will be used to read 100 bytes at a time from our file using the call to `f.Read`. This buffer will be appended to''result'' each time it's filled with `f.Read`.

And we loop. If we get an error while reading, we check if its `os.EOF` (EOF means End Of File) if so, we `break` from the loop, else it's another error so we return an empty string and that error.

At the end, we return the slice `result` converted to a string, using a type casting and `nil` which means that there was no error while retrieving the contents of the file.

I know, this example may look a little bit hard, especially since we have never used the `os` package before, but the main goal of it is to show you a use of the `defer` statement. And how it can be used to guarantee two things:

- Using `defer f.Close()` in the beginning makes sure that you will never forget to close the file, no matter how the function might change in the future, and no matter how many `return` spots will be added/removed from it.
- The close sits near the open, which is much clearer than placing it at the end of the function.

Multiple defer calls

You can defer as many function calls as you want, and they'll be executed before the containing function returns in a LIFO (Last In First Out) order. That is, the last one to be deferred will be run first, and the first one will be executed at last.

Example:

```
1   package main
2   import "fmt"
3
4   func A() {
5       fmt.Println("Running function A")
```

```
6  }
7
8  func B() {
9      fmt.Println("Running function B")
10 }
11
12 func main(){
13     defer A()
14     defer B()
15 }
```

Output:

Running function B
Running function A

And that's it for this chapter. We learned how to write variadic, and recursive functions, and how to defer function calls. The next chapter will be even more interesting, we will see how functions are, actually, values!

3.2 Even more on functions

In the previous chapter, we learned how to write variadic, and recursive functions and how to use the `defer` statement to defer functions calls to just before the currently executing function returns.

In this chapter, we will learn something that might seem surprising a little bit, which will end up being *awesome*.

3.2.1 Function types

Do you remember when we talked about *functions signatures*? That what matters most in a function, beside what it actually does, is the parameters and results numbers and types.

Well, a function type denotes the set of all functions with the same parameter and result types.

For example: the functions with signatures: func `SUM(int, int) int` and `MAX(int, int) int` are of the same type, because their parameter count, results count and types are the same.

Does this sound strange? One computes a sum, and the other the max, yet they are both of the same type?

Yes, they are. Picture this with me: The numbers 1, and 23 are both of the same type (`int`) yet their *values* are different. Paul and Sam are both of the same "human" type, yet their "values" (beliefs, jobs, weight...) are different.

The function's body, what it actually does, is what makes the difference between two functions of the same type. Just like how many coffees you can by with 1 dollar or 23 dollars.

Same thing for functions: What they do, their bodies, is what differentiate two functions of the same type.

Convinced? Let's see how to declare functions types.

How to declare a function type?

The syntax is simple:

```
type type_name func(input1 inputType1 [, input2 inputType2 [, ...])  (result1
resultType1 [, ...])
```

Some examples to illustrate this:

```
// twoOne is the type of functions with two int parameters and an int result.
// max(int, int) int and sum(int, int) int, for eg. are of this type.
type twoOne func(int, int) int

// slice_transform is the type of functions that takes a slice and outputs a
// slice.
// invert(s []int) []int and sort(s []int) []int are of this type.
type slice_transform func(s []int) []int

// varbytes is the type of variadic functions that takes bytes and outputs a
// boolean.
// redundant(...byte) bool, and contains_zero(...byte) bool are of this type
type varbytes func(...byte) bool
```

Is it clear now?

Yes, by why is this even useful or worth the hassle? You might find yourself asking. Well, I'm glad you asked this question! Read on, you will see the beautiful madness to this method!

3.2.2 Functions are values

Functions of the same type are values of this type! This means: You can declare a variable of a given function-type and assign any function of the same type to it. You can pass functions as parameters to other functions, you can also have functions that return other functions as results... And this, my friend, offers a lot of beautiful, intellignt and powerful opportunities! Let me show you some examples (don't worry, this doesn't result in Skynet... quite...).

Example 1: Write a program which when given a slice of integers will return another slice which contains only the odd elements of the first slice (yes... of course... by 'odd' I meant the 'weird' ones... what is wrong with you people?!?).

```
package main
import "fmt"

type test_int func(int) bool

// isOdd takes an ints and returns a bool set to true if the
// int parameter is odd, or false if not.
// isOdd is of type func(int) bool which is what test_int is declared to be.

func isOdd(integer int) bool {
    if integer%2 == 0 {
        return false
    }
    return true
}

// Same comment for isEven
func isEven(integer int) bool {
    if integer%2 == 0 {
        return true
    }
    return false
}

// We could've written:
// func filter(slice []int, f func(int) bool) []int
func filter(slice []int, f test_int) []int {
```

```
28      var result []int
29      for _, value := range slice {
30          if f(value) {
31              result = append(result, value)
32          }
33      }
34      return result
35  }
36
37  func main(){
38      slice := []int {1, 2, 3, 4, 5, 7}
39      fmt.Println("slice = ", slice)
40      odd := filter(slice, isOdd)
41      fmt.Println("Odd elements of slice are: ", odd)
42      even := filter(slice, isEven)
43      fmt.Println("Even elements of slice are: ", even)
44  }
```

Output:

s = [1 2 3 4 5 7]
Odd elements of s are: [1 3 5 7]
Even elements of s are: [2 4]

The functions `isOdd` and `isEven` are very simple. They both take an `int`, and return a `bool`. Of course, they're just little examples of functions of type `test_int`. One can imagine more advanced and complex functions that are of this type.

The interesting part is the `filter` function, which takes a slice of type `[]int` *and* a function of type `test_int` and returns a slice of type `[]int`.

Look at how we made a call `f(value)` on line 30. In fact, the function `filter` doesn't *care* how `f` works, what matters for it is that `f` needs an `int` parameter, and that it returns a `bool` result.

And we used this fact, in our `main` function. We called `filter` using different functions of type `test_int`.

If we're asked to extend the program to filter all the elements of the slice that, for example, are multiple of 3. All we would have to do is write a new function `is3Multiple(i int) bool` and use it with `filter`.

Now, if we want to filter the odd integers that are multiple of 3, we call filter twice likes this:

```
slice := filter(filter(s, isOdd), is3Multiple)
```

The function `filter` is good at what it does. It filters a slice given a criteria, and this criteria is expressed in the form of functions.

Cool, isn't it?

3.2.3 Anonymous functions

In the previous example, we wrote the functions `isOdd` and `isEven` outside of the main function. It is possible to declare *anonymous* functions in Go, that is functions without a name, and assign them to variables.

Let's see what this means with an example:

```
package main
import "fmt"

func main(){
    //add1 is a variable of type func(int) int
    //we don't declare the type since Go can guess it
    //by looking at the function being assigned to it.
    add1 := func(x int) int{
        return x+1
    }

    n := 6
    fmt.Println("n = ", n)
    n = add1(n)
    fmt.Println("add1(n) = ", n)
}
```

Output:

n = 6
add1(n) = 7

Our variable `add1` is assigned a *complete* function definition, minus the name. This function is said to be "anonymous" for this reason (lack of a name).

Let's see an example using anonymous functions, and that returns functions.

3.2.4 Functions that return functions

Back to the filtering problem, but now, we'd like to design it differently. We want to write a function `filter_factory` that given a single function `f` like `isOdd`, will produce a new function that takes a slice `s` of ints, and produces two slices:

- `yes`: a slice of the elements of `s` for which `f` returns `true`.
- `no`: a slice of the elements of `s` for which `f` returns `false`.

```
package main
import "fmt"

func isOdd(integer int) bool{
    if integer%2 == 0 {
        return false
    }
    return true
}

func isBiggerThan4(integer int) bool{
    if integer > 4 {
        return true
    }
    return false
}

/*
```

```
    filter_factory
    input: a criteria function of type: func(int) bool
    output: a "spliting" function of type: func(s []int) (yes, no []int)
    --
    To wrap your head around the declaration below, consider it this way:
    //declare parameter and result types
    type test_int func (int) bool
    type slice_split func([]int) ([]int, []int)
    and then declare it like this:
    func filter_factory(f test_int) slice_split
    --
    In summary: filter_factory takes a function, and creates another one
    of a completly different type.
*/
func filter_factory(f func(int) bool) (func (s []int) (yes, no []int)){
    return func(s []int) (yes, no []int){
        for _, value := range s{
            if f(value){
                yes = append(yes, value)
            } else {
                no = append(no, value)
            }
        }
        return //look, we don't have to add yes, no. They're named results.
    }
}

func main(){
    s := []int {1, 2, 3, 4, 5, 7}
    fmt.Println("s = ", s)
    odd_even_function := filter_factory(isOdd)
    odd, even := odd_even_function(s)
    fmt.Println("odd = ", odd)
    fmt.Println("even = ", even)

    //separate those that are bigger than 4 and those that are not.
    //this time in a more compact style.
    bigger, smaller := filter_factory(isBiggerThan4)(s)
    fmt.Println("Bigger than 4: ", bigger)
    fmt.Println("Smaller than or equal to 4: ", smaller)
}
```

Output:

s = [1 2 3 4 5 7]
odd = [1 3 5 7]
even = [2 4]
Bigger than 4: [5 7]
Smaller than or equal to 4: [1 2 3 4]

How does this work? First, we won't discuss `isOdd` and `isBiggerThan4` they're simple and both of the same type: `func(int) bool`. That's all that matters for us, now.

Now the heart of the program: the `filter_factory` function. Like stated in the comments, this function takes as a parameter a function `f` of type `fun(int) bool` (i.e. the same as `isOdd` and `isBiggerThan4`)

filter_factory returns an anonymous function that is of type: `func([]int)([]int, []int)` –or in plain english: a function that takes a slice of ints as its parameters and outputs two slices of ints.

This anonymous function, justly, uses `f` to decide where to *copy* each element of its input slice. if `f(value)` returns `true` then append it to the `yes` slice, else append it to the `no` slice.

Like the `filter` function in the first example, the anonymous function doesn't *care* how `f` works. All it knows is that if it gives it an `int` it will return a `bool`. And that's enough to decide where to append the value; in the `yes` or the `no` slices.

Now, back to the `main` function and how we use `filter_factory` in it.

Two ways:

The detailed one: we declare a variable `odd_even_function` and we assign to it the result of calling `filter_factory` with `isOdd` as its parameter. So odd_even_function is of type `func([]int) ([]int, []int)`.

We call `odd_even_function` with `s` as its parameters and we retrieve two slices: `odd` and `even`.

The second way is compact. The call `filter_factory(isBiggerThan4)` returns a function of type `func([]int)([]int, []int)` so we use that directly with our slice `s` and retrieve two slices: `bigger` and `smaller`.

Does it make sense, now? Re-read the code if not, it's actually simple.

3.2.5 Functions as data

Since functions have types, and can be assigned to variables, one might wonder whether it is possible to, for example, have an array or a slice of functions? Maybe a struct with a function field? Why not a map?

All this is possible in fact and, when used intelligently, can help you write simple, readable and elegant code.

Let's see a silly one:

```
package main
import "fmt"

//These consts serve as names for activities
const (
    STUDENT = iota
    DOCTOR
    MOM
    GEEK
)

//we create an "alias" of byte and name it "activity"
type activity byte

//A person has activities (a slice of activity) and can speak.
type person struct {
    activities []activity
    speak func()
}

//people is a map string:person we access persons by their names
type people map[string] person

//phrases is a map activity:string. Associates a phrase with an activity.
type phrases map[activity] string
```

```
26
27  /*
28      The function compose takes a map of available phrases
29      and constructs a function that prints a sentence with these
30      phrases that are appropriate for a "list" of the activities
31      given in the slice a
32  */
33  func compose(p phrases, a []activity) (func()){
34      return func(){
35          for key, value := range a{
36              fmt.Print(p[value])
37              if key == len(a)-1{
38                  fmt.Println(".")
39              } else {
40                  fmt.Print(" and ")
41              }
42          }
43      }
44  }
45
46  func main(){
47      // ph contains some phrases per activities
48      ph := phrases{
49          STUDENT: "I read books",
50          DOCTOR: "I fix your head",
51          MOM: "Dinner is ready!",
52          GEEK: "Look ma! My compiler works!"}
53
54      // a group of persons, and their activities
55      folks := people{
56          "Sam": person{activities:[]activity{STUDENT}},
57          "Jane": person{activities:[]activity{MOM, DOCTOR}},
58          "Rob": person{activities:[]activity{GEEK, STUDENT}},
59          "Tom": person{activities:[]activity{GEEK, STUDENT, DOCTOR}},
60      }
61
62      // Let's assign them their function "speak"
63      // depending on their activities
64      for name, value := range folks{
65          f := compose(ph, value.activities)
66          k := value.activities
67          //update the map's entry with a different person with the
68          //same activities and their function speak.
69          folks[name] = person{activities:k, speak:f}
70      }
71
72      // Now that they know what to say, let's hear them.
73      for name, value := range folks{
74          fmt.Printf("%s says: ", name)
75          value.speak()
76      }
77  }
```

Output:

Sam says: I read books.
Rob says: Look ma! My compiler works! and I read books.

Jane says: Dinner is ready! and I fix your head.
Tom says: Look ma! My compiler works! and I read books and I fix your head.

It may sound funny, and probably silly, but what I wanted to show you is how functions can be used as `struct` fields just as simply as classic data types.

Yes, I know what you are thinking; It would have been more elegant if the function `speak` of the type `person` could access her `activities` by itself and we won't even need the `compose` function then.

That's called "Object Oriented Programming" (OOP), and even though Go isn't really an OOP language, it supports some notions of it. This will be the subject of our next chapter. Stay tuned, you, geek! :)

3.3 Methods: a taste of OOP

In the previous chapter, we saw some nice things to do with functions as values that can be assigned to variables, passed to and returned from other functions. We finished with the fact that we actually can use functions as `struct` fields.

Today, we'll see a kind of an *extrapolation* of functions, that is functions with a *receiver*, which are called **methods**.

3.3.1 What is a method?

Suppose that you have a `struct` representing a rectangle. And you want *this* rectangle to *tell* you *its own* area.

The way we'd do this with functions would be something like this:

```
package main
import "fmt"

type Rectangle struct {
    width, height float64
}

func area(r Rectangle) float64 {
    return r.width*r.height
}

func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    fmt.Println("Area of r1 is: ", area(r1))
    fmt.Println("Area of r2 is: ", area(r2))
}
```

Output:

Area of r1 is: 24
Area of r2 is: 36

This works as expected, but in the example above, the function `area` is not *part* of the `Rectangle` type. It *expects* a `Rectangle` parameter as its input.

Yes, so what? You'd say. No problem, it's just if you decide to add circles and triangles and other polygons to your program, and you want to compute their areas, you'd have to write different functions *with different names* for a *functionality* or a *characteristic* that is, after all, the *same*.

You'd have to write: `area_rectangle`, `area_circle`, `area_triangle`...

And this is not elegant. Because the *area* of a *shape* is a *characteristic* of this shape. It should be a part of it, *belong* to it, just like its other fields.

And this leads us to methods: A method is function that is bound or attached to a given type. Its syntax is the same as a traditional function except that we specify a *receiver* of this type just after the keyword `func`.

In the words of Rob Pike:

> "*A method is a function with an implicit first argument, called a receiver.*"

```
func (ReceiverType r) func_name (parameters) (results)
```

Let's illustrate this with an example:

```
package main
import ("fmt"; "math") //Hey! Look how we used a semi-colon! Neat technique!

type Rectangle struct {
    width, height float64
}

type Circle struct {
    radius float64
}

/*
 Notice how we specified a receiver of type Rectangle to this method.
 Notice also how this methods -in this case- doesn't need input parameters
 because the data it need is part of its receiver r
*/
func (r Rectangle) area() float64 {
    return r.width * r.height  //using fields of the receiver
}

// Another method with the SAME name but with a different receiver.
func (c Circle) area() float64 {
    return c.radius * c.radius * math.Pi
}

func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    c1 := Circle{10}
    c2 := Circle{25}
    //Now look how we call our methods.
    fmt.Println("Area of r1 is: ", r1.area())
    fmt.Println("Area of r2 is: ", r2.area())
    fmt.Println("Area of c1 is: ", c1.area())
    fmt.Println("Area of c2 is: ", c2.area())
}
```

Output:

Area of r1 is: 24

Area of r2 is: 36
Area of c1 is: 314.1592653589793
Area of c2 is: 1963.4954084936207

A few things to note about methods:

- Methods of different receivers are different methods even if they share the name.
- A method has access to its receiver's fields (data).
- A method is called with the dot notation like `struct` fields.

So? Are methods applicable only for `struct` types? The anwser is No. In fact, you can write methods for any *named* type that you define, that is not a pointer:

```
package main
import "fmt"

//We define two new types
type SliceOfints []int
type AgesByNames map[string]int

func (s SliceOfints) sum() int {
    sum := 0
    for _, value := range s {
        sum += value
    }
    return sum
}

func (people AgesByNames) older() string {
    a := 0
    n := ""
    for key, value := range people {
        if value > a {
            a = value
            n = key
        }
    }
    return n
}

func main() {
    s := SliceOfints{1, 2, 3, 4, 5}
    folks := AgesByNames {
        "Bob": 36,
        "Mike": 44,
        "Jane": 30,
        "Popey": 100, //look at this comma. when it's the last and
    } //the brace is here, the comma above is obligatory.

    fmt.Println("The sum of ints in the slice s is: ", s.sum())
    fmt.Println("The older in the map folks is:", folks.older())
}
```

If you missed the comma remark, go back and re-read the code carefully.

Output:

The sum of ints in the slice s is: 15
The older in the map folks is: Popey

Now, wait a minute! (You say this with your best Bill Cosby's impression) *What is this "named types" thing that you're telling me now?* Sorry, my bad. I didn't need to tell you before. And I didn't want to distract you with this *detail* back then.

It's in fact easy. You can define new types as much as you want. `struct` is in fact a specific case of this syntax. Look back, we actually used this in the previous chapter!

You can create aliases for built-in and composite types with the following syntax:

```
type type_name type_literal
```

Examples:

```
//ages is an alias for int
type ages int
//money is an alias for float32
type money float32
//we define months as a map of strings and their associated number of days
type months map[string]int
//m is a variable of type months
m := months {
    "January":31,
    "February":28,
    ...
    "December":31,
}
```

See? It's actually easy, and it can be handy to give more meaning to your code, by giving names to complicated composite – or even simple – types.

Back to our methods.

So, yes, you can define methods for any named type, even if it's an alias to a pre-declared type. Needless to say that you can define as many methods, for any given named type, as you want.

Let's see a more advanced example, and we will discuss some details of it just after.

Tis the story of a set of colored boxes. They have widths, heights, depths and colors of course! We want to find out the color of the biggest box, and eventually paint them all black (Because you know... I see a red box, and I want it painted black...)

Here we *Go*!

```
package main
import "fmt"

const(
    WHITE = iota
    BLACK
    BLUE
    RED
    YELLOW
)

type Color byte

type Box struct {
```

```
    width, height, depth float64
    color Color
}

type BoxList []Box //a slice of boxes

func (b Box) Volume() float64 {
    return b.width * b.height * b.depth
}

func (b *Box) SetColor(c Color) {
    b.color = c
}

func (bl BoxList) BiggestsColor() Color {
    v := 0.00
    k := Color(WHITE) //initialize it to something
    for _, b := range bl {
        if b.Volume() > v {
            v = b.Volume()
            k = b.color
        }
    }
    return k
}

func (bl BoxList) PaintItBlack() {
    for i, _ := range bl {
        bl[i].SetColor(BLACK)
    }
}

func (c Color) String() string {
    strings := []string{"WHITE", "BLACK", "BLUE", "RED", "YELLOW"}
    return strings[c]
}

func main() {
    boxes := BoxList{
        Box{4, 4, 4, RED},
        Box{10, 10, 1, YELLOW},
        Box{1, 1, 20, BLACK},
        Box{10, 10, 1, BLUE},
        Box{10, 30, 1, WHITE},
        Box{20, 20, 20, YELLOW},
    }

    fmt.Printf("We have %d boxes in our set\n", len(boxes))
    fmt.Println("The volume of the first one is", boxes[0].Volume(), "cm³")
    fmt.Println("The color of the last one is", boxes[len(boxes)-1].color.String())
    fmt.Println("The biggest one is", boxes.BiggestsColor().String())
    //I want it painted black
    fmt.Println("Let's paint them all black")
    boxes.PaintItBlack()
    fmt.Println("The color of the second one is", boxes[1].color.String())
    //obviously, it will be... BLACK!
    fmt.Println("Obviously, now, the biggest one is", boxes.BiggestsColor().String())
}
```

Output:

```
We have 6 boxes in our set
The volume of the first one is 64 cm³
The color of the last one is YELLOW
The biggest one is WHITE
Let's paint them all black
The color of the second one is BLACK
Obviously, now, the biggest one is BLACK
```

So we defined some `const`s with consecutive values using the `iota` idiom to represent some colors.

And then we declared some types:

1. `Color` which is an alias to `byte`.
2. `Box` struct to represent a box, it has three dimensions and a color.
3. `BoxList` which is a slice of `Box`.

Simple and straightforward.

Then we wrote some methods for these types:

- `Volume()` with a receiver of type `Box` that returns the volume of the received box.
- `SetColor(c Color)` sets its receiver's color to `c`.
- `BiggestsColor()` with a receiver of type `BoxList` returns the color of the `Box` with the biggest volume that exists within the slice.
- `PaintItBlack()` with a receiver of type `BoxList` sets the colors of all `Box`es in the slice to BLACK.
- `String()` a method with a receiver of type `Color` returns a string representation of this color.

All this is simple. For real. We *translated* our vision of the problem into *things* that have methods that describe and implement a *behavior*.

3.3.2 Pointer receivers

Now, look at line 25 that I highlighted on purpose. The receiver is a pointer to Box! Yes, you can use `*Box` too. The restriction with methods is that the type `Box` itself (or any receiver's type) shouldn't be a pointer.

Why did we use a pointer? You have 10 seconds to think about it, and then read on the next paragraph. I'll start counting:

> 10, 9, 8...

Ready?

Ok! Let's find out if you were correct. We used a pointer because we needed the `SetColor` method to be able to change the value of the field 'color' of its receiver. If we did not use a pointer, then the method would recieve a *copy* of the receiver `b` (passed by value) and hence the changes that it will make will affect the copy only, not the original.

Think of the receiver as a parameter that the method has in input, and ensure that you understand and remember the difference between *passing by value and reference*.

Structs and pointers simplification

Again with the method `SetColor`, intelligent readers (You are one of them, this I know!) would say that we should have written `(*b).color = c` instead of `b.color = c`, since we need to dereference the pointer `b` to access the field color.

This is true! In fact, both forms are accepted because Go knows that you want to access the fields of the value pointed to by the pointer (since a pointer has no notion of fields) so it assumes that you wanted `(*b)` and it simplifies this for you. Look Ma, Magic!

Even more simplification

Experienced readers will also say:

> "On line 43 where we call `SetColor` on `bl[i]`, shouldn't it be `(&bl[i]).SetColor(BLACK)` instead? Since `SetColor` expects a pointer of type `*Box` and not a value of type `Box`?"

This is also quite true! Both forms are accepted. Go automatically does the conversion for you because it *knows* what type the method expects as a receiver.

In other words:

> If a method `M` expects a receiver of type `*T`, you can call the method on a variable `V` of type `T` without passing it as `&V` to `M`.

Similarly:

> If a method `M` expects a receiver of type `T`, you can call the method on a variable `P` of type `*T` without passing it as `*P` to `M`.

Example:

```
package main
import "fmt"

type Number int

//method inc has a receiver of type pointer to Number
func (n *Number) inc() {
    *n++
}

//method print has a receiver of type Number
func (n Number) print() {
    fmt.Println("The number is equal to", n)
}

func main() {

    i := Number(10) //say that i is of type Number and is equal to 10
    fmt.Println("i is equal to", i)

    fmt.Println("Let's increment it twice")
    i.inc() //same as (&i).inc() method expects a pointer, but that's okay
    fmt.Println("i is equal to", i)
    (&i).inc() //this also works as expected
    fmt.Println("i is equal to", i)

    p := &i //p is a pointer to i

```

```
    fmt.Println("Let's print it twice")
    p.print() //same as (*p).print() method expects a value, but that's okay
    i.print() //this also works as expected
}
```

Output:

i is equal to 10
Let's increment it twice
i is equal to 11
i is equal to 12
Let's print it twice
The number is equal to 12
The number is equal to 12

So don't worry, Go knows the type of a receiver, and knowing this it simplifies by accepting `V.M()` as a shorthand of `(&V).M()` and `P.M()` as a shorthand for `(*P).M()`.

Anyone who has done much C/C++ programming will have realized at this point the world of pain that Go saves us from by simply using sane assumptions.

Well, well, well... I know these pointers/values matters hurt heads, take a break, go out, have a good coffee, and in the next chapter we will see some cool things to do with methods.

3.4 More meth(odes) please!

We have learned how to write methods in Go, that they can be seen as functions with an implicit first parameter that is its receiver, and we saw some facilities that Go offers when working with pointers and receivers.

In this chapter we will see how to implement some OOP concepts using methods. It will be fun and relatively easy.

3.4.1 Structs with anonymous fields

I didn't tell you about this when we studied `structs`, but we actually can declare fields without specifying names for them (just the type). It is for this reason that we call them *anonymous fields* (the lack of a name).

When an anonymous field is a `struct` its fields are inserted (embedded) into the struct containing it.

Let's see an example to help clarify this concept:

```
package main
import "fmt"

type Human struct {
    name string
    age int
    weight int //in lb, mind you
}

type Student struct {
    Human //an anonymous field of type Human
    speciality string
```

```
13 }
14
15 func main() {
16     //Mark is a Student, notice how we declare a literal of this type:
17     mark := Student{Human{"Mark", 25, 120}, "Computer Science"}
18     //Now let's access the fields:
19     fmt.Println("His name is", mark.name)
20     fmt.Println("His age is", mark.age)
21     fmt.Println("His weight is", mark.weight)
22     fmt.Println("His speciality is", mark.speciality)
23     //Change his speciality
24     mark.speciality = "AI"
25     fmt.Println("Mark changed his speciality")
26     fmt.Println("His speciality is", mark.speciality)
27     //change his age. He got older
28     fmt.Println("Mark become old")
29     mark.age = 46
30     fmt.Println("His age is", mark.age)
31     //Mark gained some weight as time goes by
32     fmt.Println("Mark is not an athlet anymore")
33     mark.weight += 60
34     fmt.Println("His weight is", mark.weight)
35 }
```

Output:

His name is Mark
His age is 25
His weight is 120
His speciality is Computer Science
Mark changed his speciality
His speciality is AI
Mark become old
His age is 46
Mark is not an athlet anymore
His weight is 180

On line 29 and 33 we were able to access and change the fields `age` and `weight` just as if they were *declared* as fields of the `Student` struct. This is because fields of `Human` are embedded in the struct `Student`.

Cool, isn't it? But there's more than this interpolation of fields, an anonymous field can be used with its type as its name!

This means that `Student` has a field named `Human` that can be used as regular field with this name.

```
1 // We can access and modify the Human field using the embedded struct name:
2 mark.Human = Human{"Marcus", 55, 220} // Nervous breakdown, he became a Nun.
3 mark.Human.age -= 1 // Soon he will be back in diapers...
```

3.4.2 Anonymous fields of any type

Accessing, and changing an anonymous field by name is quite useful for anonymous fields that are not `structs`. * – Who told you that anonymous fields must be* `structs`?! * – I didn't!*

In fact, any *named type* and pointers to named types are completely acceptable.

Let's see some anonymous action:

```
package main
import "fmt"

type Skills []string

type Human struct {
    name string
    age int
    weight int //in lb, mind you
}

type Student struct {
    Human //an anonymous field of type Human
    Skills //anonymous field for his skills
    int //we will use this int as an anonymous field for his preferred number
    speciality string
}

func main() {
    //Jane is a Student, look how we declare a literal of this type
    //by specifying only some of its fields. We saw this before
    jane := Student{Human: Human{"Jane", 35, 100}, speciality:"Biology"}
    //Now let's access the fields:
    fmt.Println("Her name is", jane.name)
    fmt.Println("Her age is", jane.age)
    fmt.Println("Her weight is", jane.weight)
    fmt.Println("Her speciality is", jane.speciality)
    //Let's change some anonymous fields
    jane.Skills = []string{"anatomy"}
    fmt.Println("Her skills are", jane.Skills)
    fmt.Println("She acquired two new ones")
    jane.Skills = append(jane.Skills, "physics", "golang")
    fmt.Println("Her skills now are", jane.Skills)
    //her preferred number, which is an int anonymous field
    jane.int = 3
    fmt.Println("Her preferred number is", jane.int)
}
```

Output:

Her name is Jane
Her age is 35
Her weight is 100
Her speciality is Biology
Her skills are [anatomy]
She acquired two new ones
Her skills now are [anatomy physics golang]
Her preferred number is 3

The anonymous field mechanism lets us *inherit* some (or even all) of the implementation of a given type from another type or types.

3.4.3 Anonymous fields conflicts

What happens if `Human` has a field `phone` and `Student` has also a field with the same name?

This kind of conflicts are solved in Go simply by saying that the *outer* name *hides* the inner one. i.e. when accessing `phone` you are working with `Student`'s one.

This provides a way to *override* a field that is present in the "inherited" anonymous field by the one that we explicitely specify in our type.

If you still need to access the anonymous one, you'll have to use the type's name syntax:

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string //his own mobile number
}

type Employee struct {
    Human //an anonymous field of type Human
    speciality string
    phone string //work's phone number
}

func main() {
    Bob := Employee{Human{"Bob", 34, "777-444-XXXX"}, "Designer", "333-222"}
    fmt.Println("Bob's work phone is: " Bob.phone)
    //Now we need to specify Human to access Human's phone
    fmt.Println("Bob's personal phone is: " Bob.Human.phone)
}
```

Output:

Bob's work phone is: 333-222
Bob's personal phone is: 777-444-XXXX

Great! So now you ask,

> "*Now, what does all this have to do with methods?*"

Attaboy, you didn't forget the main subject of the chapter. Kudos.

3.4.4 Methods on anonymous fields

The good surprise is that methods behave exactly like anonymous fields. If an anonymous field implements a given method, this method will be available for the type that is using this anonymous field.

If the `Human` type implements a method `SayHi()` that prints a greeting, this same method is available for both `Student` and `Employee`! You won't need to write it twice for them:

```
package main
import "fmt"

type Human struct {
```

```
    name string
    age int
    phone string //his own mobile number
}

type Student struct {
    Human //an anonymous field of type Human
    school string
}

type Employee struct {
    Human //an anonymous field of type Human
    company string
}

//A human method to say hi
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func main() {
    mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
    sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}

    mark.SayHi()
    sam.SayHi()
}
```

Output:

Hi, I am Mark you can call me on 222-222-YYYY
Hi, I am Sam you can call me on 111-888-XXXX

Philosophy

Think about it: you often need to represent *things* that share some *attributes* or *characteristics*, this is solved with the anonymous field mechanism. And when you need to represent a shared *behavior* or *functionality*, you can use a method on an anonymous field.

3.4.5 Overriding a method

What if you want the `Employee` to tell you where he works as well? Easy, the same rule for overriding fields on conflicts applies for methods, and we happily exploit this fact:

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string
}
```

```
9
10 type Student struct {
11     Human //an anonymous field of type Human
12     school string
13 }
14
15 type Employee struct {
16     Human //an anonymous field of type Human
17     company string
18 }
19
20 //A human method to say hi
21 func (h *Human) SayHi() {
22     fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
23 }
24
25 //Employee's method overrides Human's one
26 func (e *Employee) SayHi() {
27     fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
28         e.company, e.phone) //Yes you can split into 2 lines here.
29 }
30
31 func main() {
32     mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
33     sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}
34
35     mark.SayHi()
36     sam.SayHi()
37 }
```

Output:

Hi, I am Mark you can call me on 222-222-YYYY
Hi, I am Sam, I work at Golang Inc. Call me on 111-888-XXXX

And... It worked like a charm!

With these simple concepts, you can now design shorter, nicer and expressive programs. In the next chapter we will learn how to enhance this experience even more with a new notion: Interfaces.

3.5 Interfaces: the awesomesauce of Go

Let's recapitulate a little bit. We saw basic data types, and using them, we learned how to create composite data types. We also saw basic control structures. We then combined all of this to write functions.

We next discovered that functions are actually a kind of data, they are themselves values and have types. We went on to learn about methods. We used methods to write functions that act on data, and then moved on to data types that *implement* a functionality.

In this chapter, we will Go to the next realm. We will now learn to use the innate spiritual ability of objects to actually *do* something.

Enough spirituality, let's make this real.

3.5.1 What is an interface?

Stated simply, interfaces are *sets of methods*. We use an interface to specify a behavior of a given object.

For example, in the previous chapter, we saw that both `Student` and `Employee` can `SayHi`, they do it differently, but it doesn't matter. They both *know* how to say "Hi".

Let's extrapolate: `Student` and `Employee` implement another method `Sing` and `Employee` implements `SpendSalary` while `Student` implements `BorrowMoney`.

So: `Student` implements: `SayHi`, `Sing` and `BorrowMoney` and `Employee` implements: `SayHi`, `Sing` and `SpendSalary`.

These sets of methods are *interfaces* that `Student` and `Employee` *satisfy*. For example: both `Student` and `Employee` satisfy the interface that contains: `SayHi` and `Sing`. But `Employee` doesn't satisfy the interafeca that is composed of `SayHi`, `Sing` and `BorrowMoney` because `Employee` doesn't implement `BorrowMoney`.

3.5.2 The interface type

An *interface type* is the specification of a set of methods implemented by some other objects. We use the following syntax:

```
type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human //an anonymous field of type Human
    school string
    loan float32
}

type Employee struct {
    Human //an anonymous field of type Human
    company string
    money float32
}

// A human likes to stay... err... *say* hi
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

// A human can sing a song, preferrably to a familiar tune!
func (h *Human) Sing(lyrics string) {
    fmt.Println("La la, la la la, la la la la la...", lyrics)
}

// A Human man likes to guzzle his beer!
func (h *Human) Guzzle(beerStein string) {
    fmt.Println("Guzzle Guzzle Guzzle...", beerStein)
}

// Employee's method for saying hi overrides a normal Human's one
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
```

```
        e.company, e.phone) //Yes you can split into 2 lines here.
}

// A Student borrows some money
func (s *Student) BorrowMoney(amount float32) {
    loan += amount // (again and again and...)
}

// An Employee spends some of his salary
func (e *Employee) SpendSalary(amount float32) {
    e.money -= amount // More vodka please!!! Get me through the day!
}

// INTERFACES
type Men interface {
    SayHi()
    Sing(lyrics string)
    Guzzle(beerStein string)
}

type YoungChap interface {
    SayHi()
    Sing(song string)
    BorrowMoney(amount float32)
}

type ElderlyGent interface {
    SayHi()
    Sing(song string)
    SpendSalary(amount float32)
}
```

As you can see, an interface can be satisfied (or implemented) by an arbitrary number of types. Here, the interface `Men` is implemented by both `Student` and `Employee`.

Also, a type can implement an arbitrary number of interfaces, here, `Student` implements (or satisfies) both `Men` and `YoungChap` interfaces, and `Employee` satisfies `Men` and `ElderlyGent`.

And finally, every type implements the *empty interface* and that contains, you guessed it: *no methods*. We declare it as `interface{}` (Again, notice that there are no methods in it.)

You may find yourself now saying:

> "*Cool, but what's the point in defining interfaces types? Is it just to describe how types behave, and what they have in common?*"

But wait! There's more! —Of course there is more!

Oh, by the way, did I mention that the empty interface has no methods yet?

3.5.3 Interface values

Since interfaces are themselves types, you may find yourself wonderng what exactly are the values of an interface type.

Tada! Here comes the wonderful news: If you declare a variable of an interface, it may store any value type that implements the methods declared by the interface!

That is, if we declare `m` of interface `Men`, it may store a value of type `Student` or `Employee`, or even... (gasp) `Human`! This is because of the fact that they *all* implement methods specified by the `Men` interface.

Reason with me now: if `m` can store values of these different types, we can easily declare a slice of type `Men` that will contain heterogeneous values. This was not even possible with slices of classical types!

An example should help to clarify what we are saying here:

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human //an anonymous field of type Human
    school string
    loan float32
}

type Employee struct {
    Human //an anonymous field of type Human
    company string
    money float32
}

//A human method to say hi
func (h Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

//A human can sing a song
func (h Human) Sing(lyrics string) {
    fmt.Println("La la la la...", lyrics)
}

//Employee's method overrides Human's one
func (e Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
        e.company, e.phone) //Yes you can split into 2 lines here.
}

// Interface Men is implemented by Human, Student and Employee
// because it contains methods implemented by them.
type Men interface {
    SayHi()
    Sing(lyrics string)
}

func main() {
    mike := Student{Human{"Mike", 25, "222-222-XXX"}, "MIT", 0.00}
    paul := Student{Human{"Paul", 26, "111-222-XXX"}, "Harvard", 100}
    sam := Employee{Human{"Sam", 36, "444-222-XXX"}, "Golang Inc.", 1000}
    Tom := Employee{Human{"Sam", 36, "444-222-XXX"}, "Things Ltd.", 5000}

    //a variable of the interface type Men
```

```
52      var i Men
53
54      //i can store a Student
55      i = mike
56      fmt.Println("This is Mike, a Student:")
57      i.SayHi()
58      i.Sing("November rain")
59
60      //i can store an Employee too
61      i = Tom
62      fmt.Println("This is Tom, an Employee:")
63      i.SayHi()
64      i.Sing("Born to be wild")
65
66      //a slice of Men
67      fmt.Println("Let's use a slice of Men and see what happens")
68      x := make([]Men, 3)
69      //These elements are of different types that satisfy the Men interface
70      x[0], x[1], x[2] = paul, sam, mike
71
72      for _, value := range x {
73          value.SayHi()
74      }
75  }
```

Output:

This is Mike, a Student:
Hi, I am Mike you can call me on 222-222-XXX
La la la la... November rain
This is Tom, an Employee:
Hi, I am Sam, I work at Things Ltd.. Call me on 444-222-XXX
La la la la... Born to be wild
Let's use a slice of Men and see what happens
Hi, I am Paul you can call me on 111-222-XXX
Hi, I am Sam, I work at Golang Inc.. Call me on 444-222-XXX
Hi, I am Mike you can call me on 222-222-XXX

As you may have noticed, interfaces types are abstract in that they don't themselves implement a given and precise data structure or method. They simply say: "if something can do *this*, it may be used *here*".

Notice that these types make *no mention* of any interface. Their implementation doesn't explicitly mention a given interface.

Similarly, an interface doesn't specify or even care which types implement it. Look how `Men` made no mention of types `Student` or `Employee`. All that matters for an interface is that if a type implements the methods it declares then it can reference values of that type.

3.5.4 The case of the empty interface

The empty interface `interface{}` doesn't contain even a single method, hence every type implements *all* 0 of its methods.

The empty interface, is not useful in *describing* a behavior (clearly, it is an entity of few words), but rather is extremely useful when we need to store *any* type value (since all types implement the empty interface).

```
// a is an empty interface variable
var a interface{}
var i int = 5
s := "Hello world"
// These are legal statements
a = i
a = s
```

A function that takes a parameter that is of type `interface{}` in actuality accepts any type. If a function returns a result of type `interface{}` then we expect it to be able to return values of any type.

How is this even useful? Read on, you'll see! (pixies)

3.5.5 Functions with interface parameters

The examples above showed us how an interface variable can store any value of any type which satisfies the interface, and gave us an idea of how we can construct containers of heterogeneous data (different types).

In the same train of thought, we may consider functions (including methods) that accept interfaces as parameters in order to be used with any type that satisfies the given interfaces.

For example: By now, you already know that fmt.Print is a variadic function, right? It accepts any number of parameters. But did you notice that sometimes, we used `strings`, and `ints`, and `floats` with it?

In fact, if you look into the fmt package, documentation you will find a definition of an interface type called Stringer

```
//The Stringer interface found in fmt package
type Stringer interface {
     String() string
}
```

Every type that implements the `Stringer` interface can be passed to `fmt.Print` which will print out the output of the type's method `String()`.

Let's try this out:

```
package main
import (
    "fmt"
    "strconv" //for conversions to and from string
)

type Human struct {
    name string
    age int
    phone string
}

//Returns a nice string representing a Human
//With this method, Human implements fmt.Stringer
func (h Human) String() string {
    //We called strconv.Itoa on h.age to make a string out of it.
    //Also, thank you, UNICODE!
    return ""+h.name+" - "+strconv.Itoa(h.age)+" years -   " +h.phone+""
}

func main() {
```

```
    Bob := Human("Bob", 39, "000-7777-XXX")
    fmt.Println("This Human is : ", Bob)
}
```

Output:

This Human is : Bob - 39 years - 000-7777-XXX

Look at how we used `fmt.Print`, we gave it `Bob`, a variable of type `Human`, and it printed it so nice and purty like! All we had to do is make the `Human` type implement a simple method `String()` that returns a string, which means that we made `Human` satisfy the interface `fmt.Stringer` and thus were able to pass it to `fmt.Print`.

Recall the *colored boxes example*? We had a type named `Color` which implemented a `String()` method. Let's again go back to that program, and instead of calling `fmt.Println` with the result of this method, we will call it directly with the color. You'll see it will work as expected.

```
//These two lines do the same thing
fmt.Println("The biggest one is", boxes.BiggestsColor().String())
fmt.Println("The biggest one is", boxes.BiggestsColor())
```

Cool, isn't it?

Another example that will make you *love* interfaces is the sort package which provides functions to sort slices of type `int`, `float`, `string` and wild and wonderous things.

Let's first see a basic example, and then I'll show you the magic of this package.

```
package main
import (
    "fmt"
    "sort"
)

func main() {
    // list is a slice of ints. It is unsorted as you can see
    list := []int{1, 23, 65, 11, 0, 3, 233, 88, 99}
    fmt.Println("The list is: ", list)

    // let's use Ints function that comes in sort
    // Ints([]int) sorts its parameter in ibcreasing order. Go read its doc.
    sort.Ints(list)
    fmt.Println("The sorted list is: ", list)
}
```

Output:

The list is: [1 23 65 11 0 3 233 88 99]
The sorted list is: [0 1 3 11 23 65 88 99 233]

Simple and does the job. But what I wanted to show you is even more beautiful.

In fact, the sort package defines a interface simply called `Interface` that contains three methods:

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less returns whether the element with index i should sort
    // before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

From the sort Interface doc:

> "A type, typically a collection, that satisfies sort. Interface can be sorted by the routines in this package. The methods require that the elements of the collection be enumerated by an integer index.

So all we need in order to sort slices of any type is to implement these three methods! Let's give it a try with a slice of `Human` that we want to sort based on their ages.

```
package main
import (
    "fmt"
    "strconv"
    "sort"
)

type Human struct {
    name string
    age int
    phone string
}

func (h Human) String() string {
    return "(name: " + h.name + " - age: "+strconv.Itoa(h.age)+ " years)"
}

type HumanGroup []Human //HumanGroup is a type of slices that contain Humans

func (g HumanGroup) Len() int {
    return len(g)
}

func (g HumanGroup) Less(i, j int) bool {
    if g[i].age < g[j].age {
        return true
    }
    return false
}

func (g HumanGroup) Swap(i, j int){
    g[i], g[j] = g[j], g[i]
}

func main(){
    group := HumanGroup{
        Human{name:"Bart", age:24},
        Human{name:"Bob", age:23},
        Human{name:"Gertrude", age:104},
        Human{name:"Paul", age:44},
        Human{name:"Sam", age:34},
```

```
        Human{name: "Jack", age: 54},
        Human{name: "Martha", age: 74},
        Human{name: "Leo", age: 4},
    }

    //Let's print this group as it is
    fmt.Println("The unsorted group is:")
    for _, v := range group {
        fmt.Println(v)
    }

    //Now let's sort it using the sort.Sort function
    sort.Sort(group)

    //Print the sorted group
    fmt.Println("\nThe sorted group is:")
    for _, v := range group {
        fmt.Println(v)
    }
}
```

Output:

The unsorted group is:
(name: Bart - age: 24 years)
(name: Bob - age: 23 years)
(name: Gertrude - age: 104 years)
(name: Paul - age: 44 years)
(name: Sam - age: 34 years)
(name: Jack - age: 54 years)
(name: Martha - age: 74 years)
(name: Leo - age: 4 years)

The sorted group is:
(name: Leo - age: 4 years)
(name: Bob - age: 23 years)
(name: Bart - age: 24 years)
(name: Sam - age: 34 years)
(name: Paul - age: 44 years)
(name: Jack - age: 54 years)
(name: Martha - age: 74 years)
(name: Gertrude - age: 104 years)

Victory! It worked like promised. We didn't implement the sorting of `HumanGroup`, all we did is implement some methods (`Len`, `Less`, and `Swap`) that the `sort.Sort` function needed to use to sort the group for us.

I know that you are curious, and that you wonder how this kind of *magic* works. It's actually simple. The function Sort of the package sort has this signature: `func Sort(data Interface)`

It accepts any value of any type that implements the interface `sort.Interface`, and deep inside (specifically: in functions that `sort.Sort` calls) there are calls for the `Len`, `Less`, and `Swap` methods that you defined for your type.

Go and look into the sort package source code, you'll see that in plain Go.

We have seen different examples of functions that use interfaces as their parameters offering an abstract way to do this on variables of different types provided they implement the interface.

Let's do that ourselves! Let's write a function that accepts an interface as it's parameter and see how brilliant we can be.

Our own example

We've seen the `Max(s []int) int` function, do you remember? We also saw `Older(s []Person) Person`. They both have the same functionality. In fact, retrieving the Max of a slice of ints, a slice of floats, or the older person in a group comes down to the same thing: loop and compare values.

Let's do it.

```
package main
import (
    "fmt"
    "strconv"
)

//A basic Person struct
type Person struct {
    name string
    age int
}

//Some slices of ints, floats and Persons
type IntSlice []int
type Float32Slice []float32
type PersonSlice []Person

type MaxInterface interface {
    // Len is the number of elements in the collection.
    Len() int
    //Get returns the element with index i in the collection
    Get(i int) interface{}
    //Bigger returns whether the element at index i is bigger that the j one
    Bigger(i, j int) bool
}

//Len implementation for our three types
func (x IntSlice) Len() int {return len(x)}
func (x Float32Slice) Len() int {return len(x)}
func (x PersonSlice) Len() int {return len(x)}

//Get implementation for our three types
func(x IntSlice) Get(i int) interface{} {return x[i]}
func(x Float32Slice) Get(i int) interface{} {return x[i]}
func(x PersonSlice) Get(i int) interface{} {return x[i]}

//Bigger implementation for our three types
func (x IntSlice) Bigger(i, j int) bool {
    if x[i] > x[j] { //comparing two int
        return true
    }
    return false
```

```
}

func (x Float32Slice) Bigger(i, j int) bool {
    if x[i] > x[j] { //comparing two float32
        return true
    }
    return false
}

func (x PersonSlice) Bigger(i, j int) bool {
    if x[i].age > x[j].age { //comparing two Person ages
        return true
    }
    return false
}

//Person implements fmt.Stringer interface
func (p Person) String() string {
    return "(name: " + p.name + " - age: "+strconv.Itoa(p.age)+ " years)"
}

/*
 Returns a bool and a value
 - The bool is set to true if there is a MAX in the collection
 - The value is set to the MAX value or nil, if the bool is false
*/
func Max(data MaxInterface) (ok bool, max interface{}) {
    if data.Len() == 0{
        return false, nil //no elements in the collection, no Max value
    }
    if data.Len() == 1{ //Only one element, return it alongside with true
        return true, data.Get(1)
    }
    max = data.Get(0)//the first element is the max for now
    m := 0
    for i:=1; i<data.Len(); i++ {
        if data.Bigger(i, m) { //we found a bigger value in our slice
            max = data.Get(i)
            m = i
        }
    }
    return true, max
}

func main() {
    islice := IntSlice{1, 2, 44, 6, 44, 222}
    fslice := Float32Slice{1.99, 3.14, 24.8}
    group := PersonSlice{
        Person{name:"Bart", age:24},
        Person{name:"Bob", age:23},
        Person{name:"Gertrude", age:104},
        Person{name:"Paul", age:44},
        Person{name:"Sam", age:34},
        Person{name:"Jack", age:54},
        Person{name:"Martha", age:74},
        Person{name:"Leo", age:4},
    }

```

```
101     //Use Max function with these different collections
102     _, m := Max(islice)
103     fmt.Println("The biggest integer in islice is :", m)
104     _, m = Max(fslice)
105     fmt.Println("The biggest float in fslice is :", m)
106     _, m = Max(group)
107     fmt.Println("The oldest person in the group is:", m)
108 }
```

Output:

The biggest integer in islice is : 222
The biggest float in fslice is : 24.8
The oldest person in the group is: (name: Gertrude - age: 104 years)

The `MaxInterface` interface contains three methods that must be implemented by types to satisfy it.

- `Len() int`: must return the length of the collection.
- `Get(int i) interface{}`: must return the element at index `i` in the collection. Notice how this method returns a result of type `interface{}` - We designed it this way, so that collections of any type may return a value their own type, which may then be stored in an empty interface result.
- `Bigger(i, j int) bool`: This methods returns `true` if the element at index `i` of the collection is bigger than the one at index `j`, or `false` otherwise.

Why these three methods?

- `Len() int`: Because no one said that collections must be slices. Imagine a complex data structure that has its own definition of length.
- `Get(i int) interface{}`: again, no one said we're working with slices. Complex data structures may store and index their elements in a more complex fashion than `slice_or_array[index]`.
- `Bigger(i, j int) bool`: comparing numbers is obvious, we know which one is bigger, and programming languages come with numeric comparison operators such <, ==, > and so on... But this notion of bigger value can be subtile. Person A is older (They say *bigger* in many languages) than B, if his age is bigger (greater) than B's age.

The different implementations of these methods by our types are fairly simple and straightforward.

The heart of the program is the function: `Max(data MaxInterface) (ok bool, max interface{})` which takes as a parameter `data` of type `MaxInterface`. `data` has the three methods we discussed above. `Max()` returns two results: a `bool` (`true` if there is a Max, `false` if the collection is empty) and a value of type `interface{}` which stores the Maximum value of the collection.

Notice how the function `Max()` is implemented: no mention is made of any specific collection type, everything it uses comes from the interface type. This *abstraction* is what makes `Max()` usable by any type that implements `MaxInterface`.

Each time we call `Max()` on a given data collection of a given type, it calls these methods as they are implemented by that type. This works like a charm. If we need to find out the max value of any collection, we only need implement the MaxInterface for that type and it will work, just as it worked for `fmt.Print` and `sort.Sort`.

That wrapps up this chapter. I'll stop here, take a break and have a good day. Next, we will see some little details about interfaces. Don't worry! It will be easier than this one, I promise.

C'mon, admit it! You've got a *NERDON* after this chapter!!!

3.6 More on interfaces

We discovered the magic of interfaces in the previous chapter, how a simple concept as defining a type's behavior can offer tremendous opportunities.

If you haven't read the previous chapter, you shouldn't read this one, simply because most of what we're about to study depends on a good understanding of the previous chapter.

Shall we begin?

3.6.1 Knowing what is stored in a interface variable

We know that a variable of a given interface can store any value of any type that implements this interface. That's the good part, but what if we wanted to retrieve a value stored in an interface variable and put it in a regular type variable, how do we know what exact type was "wrapped" in that interface variable?

Let's see an example to clarify the actual question:

```
type Element interface{}
type List []Element

//...

func main() {
    //...
    var number int
    element := list[index]
    // The question is how do I convert 'element' to int, in order to assign
    // it to number and is the value boxed in 'element' actually an int?
    //...
}
```

So, the question confronting us is:

> How do we test the type that is stored in an interface variable?

Comma-ok type assertions

Go comes with a handy syntax to know whether it is possible to convert an interface value to a given type value, it's as easy as this: `value, ok = element.(T)`, where `value` is a variable of type `T`, `ok` is a boolean, and `element` is the interface variable.

If it is possible to convert `element` to type `T`, then `ok` is set to `true`, and `value` is set to the result of this conversion. Otherwise, `ok` is set to `false` and `value` is set to the *zero value* of `T`.

Let's use this comma-ok type assertion in an example:

```
package main

import (
    "fmt"
    "strconv" //for conversions to and from string
)

type Element interface{}
type List []Element

```

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

//For printng. See previous chapter.
func (p Person) String() string {
    return "(name: " + p.name + " - age: "+strconv.Itoa(p.age) + " years)"
}

func main() {
    list := make(List, 3)
    list[0] = 1 // an int
    list[1] = "Hello" // a string
    list[2] = Person{"Dennis", 70}

    for index, element := range list {
        if value, ok := element.(int); ok {
            fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
        } else if value, ok := element.(string); ok {
            fmt.Printf("list[%d] is a string and its value is %s\n", index, value)
        } else if value, ok := element.(Person); ok {
            fmt.Printf("list[%d] is a Person and its value is %s\n", index, value)
        } else {
            fmt.Printf("list[%d] is of a different type\n", index)
        }
    }
}
```

Output:

list[0] is an int and its value is 1
list[1] is a string and its value is Hello
list[2] is a Person and its value is (name: Dennis - age: 70 years)

It's that simple!

Notice the syntax we used with our `if`s? I hope that you still remember that you can initialize inside an `if`! Do you??

Yes, I know, the more we test for other types, the more the `if`/`else` chain gets harder to read. And this is why they invented the *type switch*!

The type switch test

Better to show it off with an example, right? Ok, let's rewrite the previous example. Here we Go!

```
package main

import (
    "fmt"
    "strconv" //for conversions to and from string
)

type Element interface{}
type List [] Element
```

```
10
11 type Person struct {
12     name string
13     age int
14 }
15
16 //For printng. See previous chapter.
17 func (p Person) String() string {
18     return "(name: " + p.name + " - age: "+strconv.Itoa(p.age)+ " years)"
19 }
20
21 func main() {
22     list := make(List, 3)
23     list[0] = 1 //an int
24     list[1] = "Hello" //a string
25     list[2] = Person{"Dennis", 70}
26
27     for index, element := range list{
28         switch value := element.(type) {
29             case int:
30                 fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
31             case string:
32                 fmt.Printf("list[%d] is a string and its value is %s\n", index, value)
33             case Person:
34                 fmt.Printf("list[%d] is a Person and its value is %s\n", index, value)
35             default:
36                 fmt.Println("list[%d] is of a different type", index)
37         }
38     }
39 }
```

Output:

list[0] is an int and its value is 1
list[1] is a string and its value is Hello
list[2] is a Person and its value is (name: Dennis - age: 70 years)

Now repeat after me:

> "The `element.(type)` construct SHOULD NOT be used outside of a switch statement! – Can you use it elsewhere? – **NO, YOU CAN NOT!**"

If you need to make a single test, use the comma-ok test. Just DON'T use `element.(type)` outside of a switch statement.

3.6.2 Embedding interfaces

What's really nice with Go is the *logic* side of its syntax. When we learnt about anonymous fields in `structs` we found it quite natural, didn't we? Now, by applying the same logic, wouldn't it be nice to be able to embed an interface `interface1` within another interface `interface2` so that `interface2` "inherits" the methods in `interface1`?

I say "logic", because after all: interfaces are sets of methods, just like structs are sets of fields. And so it is! In Go you can put an interface type into another.

Example: Suppose that you have an indexed collections of elements, and that you want to get the minimum, and the maximum value of this collection without changing the elements order in this collection.

One silly but illustrative way to do this is by using the `sort.Interface` we saw in the previous chapter. But then again, the function `sort.Sort` provided by the sort package actually changes the input collection!

We add two methods: `Get` and `Copy`:

```
package main

import (
    "fmt"
    "strconv"
    "sort"
)

type Person struct {
    name string
    age int
    phone string
}

type MinMax interface {
    sort.Interface
    Copy() MinMax
    Get(i int) interface{}
}

func (h Person) String() string {
    return "(name: " + h.name + " - age: "+strconv.Itoa(h.age)+ " years)"
}

type People []Person // People is a type of slices that contain Persons

func (g People) Len() int {
    return len(g)
}

func (g People) Less(i, j int) bool {
    if g[i].age < g[j].age {
        return true
    }
    return false
}

func (g People) Swap(i, j int) {
    g[i], g[j] = g[j], g[i]
}

func (g People) Get(i int) interface{} {return g[i]}

func (g People) Copy() MinMax {
    c := make(People, len(g))
    copy(c, g)
    return c
}

func GetMinMax(C MinMax) (min, max interface{}) {
    K := C.Copy()
```

```
    sort.Sort(K)
    min, max = K.Get(0), K.Get(K.Len()-1)
    return
}

func main() {
    group := People {
        Person{name:"Bart", age:24},
        Person{name:"Bob", age:23},
        Person{name:"Gertrude", age:104},
        Person{name:"Paul", age:44},
        Person{name:"Sam", age:34},
        Person{name:"Jack", age:54},
        Person{name:"Martha", age:74},
        Person{name:"Leo", age:4},
    }

    //Let's print this group as it is
    fmt.Println("The unsorted group is:")
    for _, value := range group {
        fmt.Println(value)
    }

    //Now let's get the older and the younger
    younger, older := GetMinMax(group)
    fmt.Println("\n Younger is", younger)
    fmt.Println(" Older is ", older)

    //Let's print this group again
    fmt.Println("\nThe original group is still:")
    for _, value := range group {
        fmt.Println(value)
    }
}
```

Output:

The unsorted group is:
(name: Bart - age: 24 years)
(name: Bob - age: 23 years)
(name: Gertrude - age: 104 years)
(name: Paul - age: 44 years)
(name: Sam - age: 34 years)
(name: Jack - age: 54 years)
(name: Martha - age: 74 years)
(name: Leo - age: 4 years)

Younger is (name: Leo - age: 4 years)
Older is (name: Gertrude - age: 104 years)

The original group is still:
(name: Bart - age: 24 years)
(name: Bob - age: 23 years)

(name: Gertrude - age: 104 years)
(name: Paul - age: 44 years)
(name: Sam - age: 34 years)
(name: Jack - age: 54 years)
(name: Martha - age: 74 years)
(name: Leo - age: 4 years)

The example is idiotic (the opposite of idiomatic!) but it *did* work as desired. Mind you, interface embedding can be very useful, you can find some interface embedding in some Go packages as well. For example, the container/heap package that provides heap operations of data collections that implement this interface:

```
//heap.Interface
type Interface interface {
    sort.Interface //embeds sort.Interface
    Push(x interface{}) //a Push method to push elements into the heap
    Pop() interface{} //a Pop elements that pops elements from the heap
}
```

Another example is the io.ReadWriter interface that is a combination of two interfaces: io.Reader and io.Writer both also defined in the `io` package:

```
// io.ReadWriter
type ReadWriter interface {
    Reader
    Writer
}
```

Types that implement io.ReadWriter can read and write since they implement both Reader and Writer interfaces.

Did I mention yet to make sure you *never* use `element.(type)` outside of a switch statement?

CHAPTER 4

Like a maestro with his orchestra

Why think of programs as a sequential ballade executed by a single musician? We can do better by making every musician do what he/she has to do, and make them *communicate* with one another.

- Introduction to concurrency with Go
- Sharing and Caring (Communicating with channels)
- Channels on steroids
- ...

4.1 Colophon

This humble work was and is still being done on vim using Sphinx with the help of Graphviz (for the various drawings and diagrams) and some tea as the author's fuel.

The font used for the titles is Rancho Regular released by Sideshow under the Apache License, version 2.0

The examples provided in this book are in public domain. You can do whatever you want with them and I claim no responsibility over their use. Just leave me alone.

The whole work is produced in HTML5, and I don't care if your browser is old. It's easier for you to upgrade your browser than for me to rewrite the template used to generate this. I don't care :)

The Go gopher was drawn by Renée French, the artist who also made Glenda the Plan 9 bunny.

4.1.1 License

Your pull requests for new content, typos, better ideas... are more than welcome! If you want to translate this work, please let me know, I'd love to share your translation here too.

A little request, and I know most of you won't do it anyway. But I'd hate it if this work was copied and pasted to spam search engines and make money out of it. Really.

4.1.2 Credits

I, Big Yuuta, am the author and the maintainer of this project, and despite my weak English, I'm doing my best to keep it fun and -I hope- useful for people wanting to learn Go.

But I wouldn't be able to do this alone. In fact, I'd like to thank these people from the bottom of my heart.

- Wayne E. Seguin: This man is a real *machine*, he reviewed everything, asked questions, suggested, and corrected many sentences, gave me some great advice... And added more fun and humor to the whole thing. Wayne, you rock!
- Adam (hammerandtongs): Thank you so much for those many spelling and grammar fixes!
- Calvin (csmcanarney): Thanks for being the first to give a hand and for your fixes.
- The whole golang-nuts community for their support and cheers, and the huge amount of inspiration and ideas they share everyday in that group.

CHAPTER 5

And still be open and friendly

This is the goal of this humble work. I hope that you'll enjoy reading it, and I'd love to hear back from you! You can post issues, typos and suggestions on the github repository

5.1 Table of contents