learn-go

- Strong and statically typed:

> means that the type of a variable cannot change over time. So when you declare a variable a to hold an integer, it's always going to hold an integer, you can't put a Boolean in it, you can't put a string in it. And static typing means that all of those variables have to be defined at compile time.

- Key features:
    - simlicity
    - Fast compile times
    - Garbage collected

    > which means that you're not going to have to manage your own memory. Now you can manage your own memory. But by and large, the go runtime is going to manage that for you.

    - Built-in concurrency
    - Compile to standalone binaries

Variables

- Variable declaration
- Redeclaration and shadowing
- Visiblity
- Naming conventions
- Type conventions

```
package main

import (
        "fmt"
)

func main() {
        var i int               //var declration, case is not ready to assign a value yet
        i=45                   // var assignment
        var j int =66         //declration and assignment same line, case we need to force a type not let it to the value
  k :=10               // var assingment the declration is figured out by go compiler
        fmt.Println(i, j, k)
}
```

- at the package level you can not use k:= synatx you have to declare the variable

```
package main

import (
        "fmt"
)

var j int =66         //package level

func main() {

        fmt.Println( j)
}
```

- declare a block of variables

```
var(
        actorName string="ahmed"
        companion string="hamo"
        doctorNumber int=7
        season       int=9
)

func main() {

        fmt.Println()
}
```

- variable shadwing: function level takes the precedence on the package level for that case, but the package level still avialable for others

```
var i int=27

func main() {
        var i int=9
        fmt.Println(i)
}
```

- declared variable must be used in go

rules to declare vars

- how naming convetion can affect scope visiblity

1. lower case only visible to the package

```
package main

import (
        "fmt"
)

var i int=27

func main() {

        fmt.Println(i)
}
```

2. upper case visible to the outsider packages

```
package main

import (
        "fmt"
)

var I int=27

func main() {

        fmt.Println(i)
}
```

- the scope of the variable defines how long is the name to a var
- acronyms should be upper case like HTTP, URL

how to convert var type

- show case to convert an integer to string

1. this will output the string with 27 in the unicode not a 27 conveted to string type

```
package main

import (
        "fmt"
)

var i int=27

func main() {

        fmt.Printf("%v, %T\n", i, i)

        var j string
        j=string(i)
        fmt.Printf("%v, %T\n", j, j)
}
```

2. you have to import strconv to actually change the type of i for j

```
package main

import (
        "fmt"
        "strconv"
)

var i int=27

func main() {

        fmt.Printf("%v, %T\n", i, i)

        var j string
        j=strconv.Itoa(i)
        fmt.Printf("%v, %T\n", j, j)
}
```

arithmatic ops

```
package main

import (
        "fmt"
)


func main() {
        a :=10
        b :=3
        fmt.Printf("%v\t, %T, %T\t\n", a + b, a ,b)      // 13
        fmt.Printf("%v\t, %T, %T\t\n", a - b, a ,b)     // 7
        fmt.Printf("%v\t, %T, %T\t\n", a * b, a ,b)    // 30
        fmt.Printf("%v\t, %T, %T\t\n", a / b, a ,b)   // 3 dops the float because it the two operands are int
        fmt.Printf("%v\t, %T, %T\t\n", a % b, a ,b)  //1  pick up the remainder
}
```

if two var one is int and the second is int8 you have to type convert

```
package main

import (
        "fmt"
)


func main() {
        var a int =10
        var b int8 =3
        fmt.Println(a + int(b)) // or   fmt.Println(int8(a) + b)

}
```

wisbit operations

```
package main

import (
        "fmt"
)


func main() {
        a :=10                 // 0b1010
        b :=3                 // 0b0011

        fmt.Println(a & b)     // AND   =0010    =2
        fmt.Println(a | b)    // OR    =1011    =11
        fmt.Println(a ^ b)   //XOR    =1001    =9
        fmt.Println(a &^ b) //ANDNOT =0100    =8

}
```

shifting means 2^x : base 2 to the power x
shiftleft means multiply, shiftright means divid

```
package main

import (
        "fmt"
)


func main() {
        a :=8  //2^3=8
        fmt.Println(a<<2)  //shiftleft means multiply 2^3 * 2^2=2^5 =32
        fmt.Println(a>>3)  //shiftright mean divid   2^3 / 2^3 =2^1 =1


}
```

> shifting and remainder operators are only with intgers

complex numbers

there are two types of complex numbers. There's complex 64, and complex 128. go undersand the equations of the complex numbers

```
package main

import (
        "fmt"
)


func main() {
        var n complex64 =1 + 2i
        var m complex64 = 2i
        fmt.Printf("%v, %T\n", n, n)  // (1+2i), complex64
        fmt.Printf("%v, %T", m,m)    // (0+2i), complex64

}
```

operations with complex nubmers

```
package main

import (
        "fmt"
)


func main() {
        var n complex64 =1 + 2i
        var m complex64 = 2i
        fmt.Println( n+m)  // (1+4i)
        fmt.Println( n-m)  //(1+0i)
        fmt.Println( n*m)  //(-4+2i)
        fmt.Println( n/m)  //(1-0.5i)
}
```

destructing the complex number to get real and imagine number

```
package main

import (
        "fmt"
)


func main() {
        var n complex64 =1 + 2i
        fmt.Printf("%v , %T\n", real(n), real(n)) // 1 , float32
        fmt.Printf("%v , %T\n", imag(n), imag(n))  // 2 , float32

}
```

crate a complex number by Complex function

```
package main

import (
        "fmt"
)


func main() {
        var n complex128 =complex(5, 15)
        fmt.Printf("%v, %T\n", n, n)  //(5+15i), complex128
}
```

Texting types

- String type represent UTf-8 charactars

```
func main() {
        s := "this a string"
        fmt.Printf("%v, %T\n", s, s)
}
```

one of the interesting aspects of a string is we can actually treat it sort of like an array. treat the string of text as a collection of letters.

```
func main() {
        s := "this a string"
        fmt.Printf("%v, %T\n", s[2], s[2])  // 105, uint8
}
```

> - what the heck happened there? Well, what's happening is that strings in go are actually aliases for bytes.
> - strings are generally immutable.
> - there is one arithmetic or pseudo arithmetic operation that we can do with strings, and that is string concatenation. Or in simpler terms, we can add strings together.

```
func main() {
        s := "this a string"
        s2 :="iam the second string "
        fmt.Println( s+s2)
}
```

convert them to collections of bytes

```
func main() {
        s := "this a string"
        b := []byte(s)
        fmt.Printf("%v, %T\n", b,b)
}
```

> [116 104 105 115 32 97 32 115 116 114 105 110 103], []uint8
> we actually get this as a string comes out as the ASCII values, or the UTF valuesfor each character in that string.

> why would you use this one? It's a very good question. A lot of the functions that we're going to use in go actually work with byte slices. And that makes them much more generic and much more flexible than if we work with hard coded strings. So for example, if you want to send as a response to a web service call, if you want to send a string back, you can easily convert it to a collection of bytes. But if you want to send a file back, well, a file on your hard disk is just a collection of bytes, too. So you can work with those transparently and not have to worry about line endings and things like that. So while in your go programs, you're going to work with strings a lot as strings. When you're going to start sending them around to other applications or to other services, you're very often going to take advantage of this ability to just convert it to a byte slice.

- Rune represent UTF-32 charactars int32

Constants

- Naming convention
- Typed constants
- Untyped constants
- Enumerated constants
- Enumeration expressions

```
func main() {
        const myConst   // internal constant
        const MyConst   // global constant to be exported
  const i int =40  // named type constant
}
```

characteristic of a constant is that it has to be assignable at compile time. you can not assign a calculation of something to const

```
func main() {

        const myConst float64=math.Sin(1.57)
        fmt.Printf("%v, %T\n", myConst, myConst) // get error
}
```

Enums

what is Iota? Well, Iota is a counter that we can use when we're creating what are called enumerated constants.

```
package main

import "fmt"

const(
        a=iota
        b=iota
        c=iota
)

func main() {
        fmt.Printf("%v\n",a) //0
        fmt.Printf("%v\n",b) //1
        fmt.Printf("%v\n",c) //2

}
```

if we don't assign the value of a constant after the first one, then the compiler is going to try and figure the pattern of assignments.
that value of Iota is scoped to that constant block.
use iota as flag checking, also we can use iota to check a variable is assigned a value yet, or equal to zero value of the constant

```
package main
import "fmt"
const(
        errorSpecialist=iota
        catSpecialist
        dogSpecialist
        snakeSpecialist
)
func main() {
        var specialistType int
        fmt.Printf("%v\n",specialistType==catSpecialist)
}
```

we can use this underscore symbol if we don't care about zero, then we don't have any reason to assign the memory to it.
And basically, what that tells the compiler is yes, I know you're going to generate a value here, but I don't care what it is go ahead and throw that away.

this can be valuable if you need some kind of a fixed offset.

```
package main

import "fmt"

const(
        _ =iota +5
        catSpecialist
        dogSpecialist
        snakeSpecialist
)

func main() {
        var specialistType int
        fmt.Printf("%v\n",specialistType==catSpecialist)
        fmt.Printf("%v\n",catSpecialist)
        fmt.Printf("%v\n",dogSpecialist)
        fmt.Printf("%v\n",snakeSpecialist)

}
```

use case shifleft is essentially multiply by 2 to the power of x

```
package main

import "fmt"

const(
        _ =iota  // igonre first value by assigning to blank identifier
        KB=1<<(10*iota)
        MB
        GB
        TB
        PB
        EB
        ZB
        YB
)

func main() {
        fileSize :=4000000000.
        fmt.Printf("%.2fGB", fileSize/GB)
}
```

> - =iota is zero
>   KB=1<<0 MEANS NO SHIFTING
>   MB=1<<10 MEANS 1*2^10*
>   *GB=1<<100 MEANS 1*2^{100}
>   TB=1<<1000 MEANS 1*2_{1000}

- another use case
  let's just say that we've got an application and that application has users and those users have certain roles. So inside of this constant block, here, I'm defining various roles that we can have. So for example, you might be an admin, you might be at the headquarters or out in the field somewhere, you might be able to see the financials or see the monetary values. And then there may be some regional roles. So can you see properties in Africa, can you see properties in Asia, Europe, North America, or South America. So in order to define these constants, what I'm doing is I'm setting the value to one bit shifted iota.

So the first constant is admin is one bit shifted zero places, so it's a literal one, the second one is one bit shifted one place, that's two, and then four, and then eight, and then 16, and so on.

in the main program, I'm defining the roles in a single byte.

```
package main

import "fmt"

const(
        isAdmin=1<<iota            // 1
        isHeadquarters            //2
        canSeeFinancials         //4
        canSeeAfrica            //8
        canSeeAsia             //16
        canSeeEurope          //32
        canSeeNorthAmerica   //64
        canSeeSouthAmerica  //128

)

func main() {
        var roles byte= isAdmin | canSeeFinancials | canSeeEurope

        fmt.Printf("%b\n", roles)   // binary representation of ORing the three active roles and storing them in one variable

        // check if a user is admin or any other role to check against using bitwise bitmask mathimatics
        fmt.Printf("Is Admin? %v\n", isAdmin&roles == isAdmin)

        fmt.Printf("Can see Africa? %v", canSeeAfrica&roles == canSeeAfrica)

}
```

Arrays

- Creation
- Built-in functions
- working with array Why do we need them and what are they used for? is a very powerful way for us to work with our data. Now, another advantage that we have with working with arrays is the way that they're laid out in memory. the design of the language that these elements are continous in memory, which means accessing the various elements of the array is very, very fast.

```
package main
import "fmt"
func main() {
        // first way to declare array
        var Arr [3]int=[3]int {1,2,3}
        fmt.Printf("Arr: %v\n", Arr)
        // second way
        var array [5]int
        array[0]=2
        array[1]=3
        array[2]=4
        array[3]=8
        fmt.Printf("array:%v\n", array)
        // third way
        grades := [...]int {97,85,93}
        fmt.Printf("Grades: %v", grades)
}


func main() {
        var students [5]string
        fmt.Printf("students: %v\n", students)
        students[0]="lisa"
        students[1]="ahmed"
        students[2]="hoda"
        fmt.Printf("student #1: %v\n", students[1])
        fmt.Printf("no of Students: %v\n", len(students))
}
```

arrays of arrays

```
func main() {
        var identityMatix [3][3] int= [3][3]int{ [3]int{1,0,0}, [3]int{0,1,0}, [3]int{0,0,1} }
        // another way intialize each row indvidually
        var idMatrix [3][3] int
        idMatrix[0]=[3]int{1,0,0}
        idMatrix[1]=[3]int{1,1,1}
        idMatrix[2]=[3]int{0,0,0}
        fmt.Printf("Ids1 %v\n", idMatrix)
        fmt.Printf("ids2 %v\n", identityMatix)

}
```

arrays are actually considered values. When you copy an array, you're actually creating a literal copy So it's not pointing to the same underlying data is pointing to a different set of data, which means it's got to reassign that entire length of the array.
they have a fixed size that has to be known at compile time

if you're passing arrays into a function, go is going to copy that entire array over.
So what do you do if you don't want to have this behavior? idea of pointers.

here a, b are different sets of array after b copied the a array

```
func main() {
        a:=[...]int{1,2,3}
        b:=a
        fmt.Println(a)
        fmt.Println(b)
}
```

here b point to a array values not copying in actual new array

```
func main() {
        a:=[...]int{1,2,3}
        b:=&a
        fmt.Println(a)
        fmt.Println(b)
}
```

Slices

An array has a fixed size. A slice, on the other hand, is a dynamically-sized, flexible VIEW into the elements of an array.
we can have a very large array and only be looking at a small piece of it.
slices are naturally what are called reference types.So they refer to the same underlying data. A slice does not store any data, it just describes a section of an underlying array.

we see that A and B are actually pointing to the same underlying array. if one of those slices changes the underlying data, it could have an impact somewhere else in your application.

```
func main() {
        a:=[]int{1,2,3}
        b:=a
        b[1]=5
        fmt.Println(a)
        fmt.Println(b)
        fmt.Printf("length: %v\n", len(a))
        fmt.Printf("capacity: %v\n", cap(a))
}


func main() {
        a:=[]int{1,2,3, 4,5,6,7,8,9,10}

        b:=a[:]          // slice of all elements
        c:=a[3:]         //slice from the 4th element to end
        d:=a[:6]         // slice from 0element to 5th
        e:=a[3:6]       // slice  4th, 5th, 6th

        a[5]=45        // this modification affect all the slices

        fmt.Println(a)
        fmt.Println(b)
        fmt.Println(c)
        fmt.Println(d)
        fmt.Println(e)

}
```

capacity and length of slice according its position to the underlying array endpoint

```
package main

import "fmt"

func main() {
        a := []int{1, 2, 3, 4, 5, 6, 7}
        fmt.Println(cap(a))
        b := a[:5]
        fmt.Println(len(b))
        fmt.Println(cap(b))
        b1 := b[3:4]
        fmt.Println(b1)
        fmt.Println(len(b1)) // length of slice how many element slice has
        fmt.Println(cap(b1)) // capacity of slice how many element i can have from my start point to the underlying array end point
        // here b1 start point is 4 and the underlying array end point is 7 so cap is 4

}
```

use make function to create slice

```
package main

import "fmt"

func main() {
        // make built-in function to create a sllice take 2 or 3 args
        // here 1st is the type, the 2nd is the length of the slice, the 3rd is the capacity of the underlying array
        a := make([]int, 3)
        fmt.Println(a)
        fmt.Printf("length %v\n", len(a))
        // to add an element to slice use append func, this add number 2 to the slice
        a = append(a, 2)
        fmt.Println(a)
        fmt.Printf("length %v\n", len(a))

        //output:
        /*
                [0 0 0]
                length 3
                [0 0 0 2]
                length 4
        */

}
```

re-slice a slice to extend itself

```
package main
import "fmt"
func main() {
        a := [5]int{1, 2, 3, 4, 5}
        fmt.Println(a)
        fmt.Printf("length %v\n", len(a))
        fmt.Printf("capacity %v\n", cap(a))
        s := a[1:3]
        fmt.Println(s)
        fmt.Printf("length %v\n", len(s))
        fmt.Printf("capacity %v\n", cap(s))

        // here re-slice the s slice to extend it to the last element of the underlying array which is a
        fmt.Println(s[:cap(s)])
        fmt.Printf("capacity %v\n", cap(s))
}
```

- Important note if S slice has a certain nuber of space and we used them all, to append a new element to S, we have to copy S into double size for so the new element can be addded

```
package main
import "fmt"
func main() {
        a := [...]int{3, 5, 8, 10, 12}
        fmt.Println(a)           //[3 5 8 10 12]
        fmt.Println(cap(a))     //5

        b := a[:]

        b = append(b, 3)
        fmt.Println(b)          //[3 5 8 10 12 3]
        fmt.Println(cap(b))     //10

}
```

another example

```
package main

import "fmt"

func main() {
        a := []int{}
        fmt.Println(a)
        fmt.Printf("len:%v\n", len(a))
        fmt.Printf("cap:%v\n", cap(a))
        //[]
        //len:0
        //cap:0

        a=append(a,1)
        fmt.Println(a)
        fmt.Printf("len:%v\n", len(a))
        fmt.Printf("cap:%v\n", cap(a))
        //[1]
        //len:1
        //cap:1

        a=append(a,2,3,4,5)
        fmt.Println(a)
        fmt.Printf("len:%v\n", len(a))
        fmt.Printf("cap:%v\n", cap(a))
        //[1 2 3 4 5]
        //len:5
        //cap:6
}
```

> when the sequence is appending, here is what is happening, a double itself so the sequence 2,3,4,5 starts to pool its elements, so 2,3 append but still 4, 5 , so a double it size again which is 2 . now after all values entered the new slice is 6 capacity

- concatenate two slices togther you can't do that directly, use ... to spread the the second slice

```
package main

import "fmt"

func main() {
        var s []int = []int{9, 5, 3}

        s = append(s, []int{2, 3, 4}...)
        printSlice(s)
}

func printSlice(s []int) {
        fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

- pop elements from slice

```
package main

import "fmt"

func main() {
        var s []int = []int{9, 5, 3}
        // remove first element
        b := s[1:]
        fmt.Println(b)
        //[5 3]

        // remove last element
        z := s[:len(s)-1]
        fmt.Println(z)
        //[9 5]

}
```

remove the middle element from a slice we could append first half until middle element with second half after middle element

```
package main

import "fmt"

func main() {
        var s []int = []int{9, 5, 3, 4, 5}
        var middle = len(s) / 2

        b := append(s[:middle], s[middle+1:]...)
        fmt.Println(b)

}
```

Maps

- what arey they?
- creating
- manipulation

So what maps provides us is a very flexible data type. When we're trying to map one key type over to one value type.

```
package main

import "fmt"

func main() {
        statePopulation := map[string]int{
                "California": 39250017,
                "Texas":      27862596,
                "Florida":    20612439,
                "New York":   19745289,
                "Ohio":       11614373,
        }

        fmt.Println(statePopulation)
}
```

slice can not be a key type for maps, but arrays can

create a map use make built-in function

```
package main

import "fmt"

func main() {
        fruits := make(map[string]string)
        fruits = map[string]string{
                "orange": "orange",
                "apple":  "green",
                "banana": "yellow",
        }

        fmt.Println(fruits)
}
```

- Maps manipulation
- manipulation maps happens almost instantly, access, changing , deleting values, no matter how big the map this will happen very fast the order in the map is not provided, keys are stored with no ordering

```
package main

import "fmt"

func main() {
        statePopulation := map[string]int{
                "California": 39250017,
                "Texas":      27862596,
                "Florida":    20612439,
                "New York":   19745289,
                "Ohio":       11614373,
                "Brookln":    116143276,
        }

        // add new pairs of key and value
        statePopulation["Georgia"] = 19745289
        fmt.Println(statePopulation)

        // access value of a key from maps
        fmt.Println(statePopulation["Ohio"])

        // delete value of a key from maps
        delete(statePopulation, "Ohio")
        fmt.Println(statePopulation)

        //note, about deleting a key, the value is modified to the default value of key type zero for int, this indicates that specific
key has no value
        // to be sure that the key is not registered use the flag 'ok' to make sure the key was not registered anyway or deleted
        _, ok := statePopulation["oho"]
        fmt.Println(ok) //  false

        // length of maps
        fmt.Println(len(statePopulation))
}
```

when you have multiple assignments to map, the underlying data is passign by REFRENCE, which means, manipulating one variable that points to the map, is gonna has impact on the other ones

```
package main

import "fmt"

func main() {
        statePopulation := map[string]int{
                "California": 39250017,
                "Texas":      27862596,
                "Florida":    20612439,
                "New York":   19745289,
                "Ohio":       11614373,
                "Brookln":    116143276,
        }

        sp := statePopulation
        delete(sp, "Ohio")
        //this will delete Ohio from sp and statePopulation
        fmt.Println(sp)
        fmt.Println(statePopulation)
}
```

Struct

- Collections of disparate data types that describe a single concept
- keyed by named fields
- normally created as types, but anonymous structs are allowed
- structs are value types
- no inheritance, but can use composition via embedding
- tags can be added to struct fields to describe the field

what the struct type does is it gathers information together that are related to one concept, in this case, a doctor. And it does it in a very flexible way. Because we don't have to have any constraints on the types of data that's contained within our struct, we can mix any type of data together. And that is the true power of a struct. of the other collection types we've talked about have had to have consistent types. So arrays always have to store the same type of data slices have the same constraint. And we just talked about maps and how their keys always have to have the same type. And their values always have to have the same type within the same map.

- here is how to create a struct with three diff type. int, string, slice

```
package main

import "fmt"

type Doctor struct {
        number     int
        actorName  string
        companions []string
        episodes   []string
}

func main() {
        aDoctor := Doctor{
                number:    3,
                actorName: "ahmed khalid",
                companions: []string{
                        "hamo",
                        "hoda",
                },
        }

        fmt.Println(aDoctor)

        //acces field of struct
        fmt.Println(aDoctor.actorName)

        // access fields like a slice, e.g second element of slice
        fmt.Println(aDoctor.companions[1])

}
```

> note:
> naming rules for struct follows the same as go other variable struct starts with upper case letter in the main package is exported to other, the struct fields must be upper case also if it is required to make them accessible to other packages

> note: it is better approach to user field names syntax advantage, if I don't have any information about the episodes at this point in my program, I actually can ignore the fact that that field exists. And what this means is I changed the underlying struct without changing the usage at all, which makes my application a little bit more robust and change proof

anonymous struct

So instead of setting up a type, and saying, doctor, and that's going to be a struct, and that's going to have a single field called name, that's going to take a string. We're condensing all of that into this single declaration

```
package main

import "fmt"

func main() {
        aDoctor := struct{ name string }{name: "hamo"}
        fmt.Println(aDoctor)

}
```

when are you going to use this,in situations where you need to structure some data in a way that you don't have in a formal type. But it's normally only going to be very short lived. So you can think about if you have a data model that's coming back in a web application, and you need to send a projection or a subset of that data down to the client, you could create an anonymous struct in order to organize that information. So you don't have to create a formal type that's going to be available throughout your package for something that might be used only one time.

> unlike maps, **structs are value types** .unlike maps, and slices, these are referring to independent datasets. So when you pass a struct around in your application, you're actually passing copies of the same data around.

```
package main
import "fmt"

func main() {
        aDoctor := struct{ name string }{name: "hamo"}
        fmt.Println(aDoctor.name) // hamo
        anotherDoctor := aDoctor
        anotherDoctor.name = "hoda"
        fmt.Println(aDoctor.name) //hamo, nothing changed, because anotherDoctor is inependent sturct
}
```

just like with arrays, if we do want to point to the same underlying data, we can use that address of operator. And when we run this, we have in fact, both variables pointing to the same underlying data.

```
anotherDoctor :=&aDoctor
```

embeding in struct

go language doesn't support traditional object oriented principles.

how am I going to create my program if I don't have inheritance available? Well, let me show you what go has, instead of an inheritance model. It uses a model that's similar to inheritance called composition. So where inheritance is trying to establish the is a relationship.

So if we take this example here, if we were in a traditional object oriented language, we wouldn't want to say that a bird is an animal, and therefore a bird has a name a bird has an origin, a bird has also bird things like its speed, and if it can fly or not

it supports composition through what's called embedding. So right now we see that animal and bird are definitely independent structs, there's no relationship between them. However, I can say that a bird has animal like characteristics by embedding an animal struct

```
package main

import "fmt"

type Animal struct {
        Name   string
        Origin string
}

type Bird struct {
        Animal
        SpeedKPH float32
        CanFly   bool
}

func main() {

        b := Bird{}
        //or
        var b Bird= Birde{}
        b.Name = "Emu"
        b.Origin = "Austerial"
        b.SpeedKPH = 48
        b.CanFly = false
        fmt.Println(b)

        // or this way
        b := Bird{
                Animal: Animal{Name: "Emu", Origin: "Australia"},
                SpeedKPH:48,
                CanFly:false,
        }
        fmt.Println(b)

}
```

tag

in order to describe some specific information about this name field. So let's say for example, that I'm working with some validation framework. So let's just say that I'm working within a web application, and the user is filling out a form and two of the fields are providing the name and the origin. And I want to make sure that the name is required and doesn't exceed a maximum length

```
type Animal struct {
        Name   string ` required max:"100" `
        Origin string
}
```

- Demo to show how struct are value typed and how using pointer with them

1. we can have a bunch of different objects of the same struct with different values

```
package main

import "fmt"

type Point struct {
        x int32
        y int32
}

//take refeerenc of Point Struct to have access to it for some modificatin
func changeX(ptr *Point) {
        ptr.x = 100
}

func main() {

        p1 := &Point{x: 0, y: 4}
        fmt.Println(p1)
        changeX(p1)
        fmt.Println(p1)

}
```

- struct with metods

```
package main

import "fmt"

type Student struct {
        name   string
        grades []int
        age    int
}

// method for student struct
func (s Student) getStudentAge() int {
        return s.age
}

// use a pointer to modify student age field
func (s *Student) setStudentAge(age int) {
        s.age = age
}

func (s Student) getAverageGrades() float32 {

        sum := 0
        for _, v := range s.grades {
                sum += v
        }
        return float32(sum) / float32(len(s.grades))
}
```

```
func (s Student) getMaxGrade() int {
        max := 0
        for _, v := range s.grades {
                if max < v {
                        max = v
                }
        }
        return max
}

func main() {
        // s1 can access methods of student struct
        s1 := Student{
                name:   "hamo",
                grades: []int{60, 95, 83, 91, 82},
                age:    25,
        }

        fmt.Println(s1.getStudentAge()) //25
        s1.setStudentAge(26)
        fmt.Println(s1.getStudentAge()) //26
        average := s1.getAverageGrades()
        fmt.Println(average) //82.2

        max := s1.getMaxGrade()
        fmt.Println(max) //
}
```

Flow Control

For

Go has only one looping construct, the for loop.

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

The init and post statements are optional.

For is Go's "while"

```
func main() {
        sum := 1
        for sum < 1000 {
                sum += sum
        }
        fmt.Println(sum)
}
```

break statment

```
func main() {
        i:=0
        for{
                fmt.Println(i)
                i++
                if i==5{
                        break
                }
        }
}
```

if i equal to five, then we are gonna break out from the for loop
break, breaks out from the closest for loop in a nested for loop, but if in a certain condition is met i want to break out from the parent loop, we can use a Lable to the parent for loop,

```
func main() {
Loop:
        for i := 0; i < 10; i++ {
                for j := 0; j < 3; j++ {
                        fmt.Println(i * j)
                        if i*j >= 3 {
                                break Loop
                        }
                }
        }
}
```

> Loop, where it is put, it describe where i want to break out

loop through collections

using rang key word to loop maps, slices, arrays. and strings

```
func main() {
        s := []int{1, 3, 5, 9}
        for key, value  := range s {
                fmt.Println(k, v)
        }
}
```

loop through string

```
func main() {
        s := "hello go !"
        for k, v := range s {
                fmt.Println(k, string(v)) // values are casting to string
        }
}
```

continue

is used to skip a condition execution when it is true
like in a for loop, when looping i want to skip certain conditon but, keep looping until other iterations are finished

```
func main() {
        for i := 0; i < 10; i++ {
                if i%2 == 0 {
                        continue
                }
                fmt.Println(i)
        }
}
```

> here i want to skip even number, only print the odd numbers

If

```
func sqrt(x float64) string {
        if x < 0 {
                return sqrt(-x) + "i"
        }
        return fmt.Sprint(math.Sqrt(x))
}
```

If with a short statement

```
func pow(x, n, lim float64) float64 {
        if v := math.Pow(x, n); v < lim {
                return v
        }
        return lim
}
```

Switch

break statement that is needed at the end of each case is provided automatically in Go. Another important difference is that Go's switch cases need not be constants, and the values involved need not be integers.

Switch without a condition is the same as switch true. This construct can be a clean way to write long if-then-else chains.

a very simple way to compare one variable to multiple possible values against variable

```
func main() {
        switch 3 {
        case 1, 5, 10:
                fmt.Println("one, five, and ten")
        case 2, 4, 6:
                fmt.Println("two, four, and six")
        default:
                fmt.Println("another number")
        }
```

> be aware of is the test cases do have to be unique.

in our switch, we can use an initializer
the initializer doesn't have to generate a Boolean result,

```
func main() {
        switch i := 2 + 3; i {
        case 1, 5, 10:
                fmt.Println("one, five, and ten")
        case 2, 4, 6:
                fmt.Println("two, four, and six")
        default:
                fmt.Println("another number")
        }
}
```

- tagless syntax is arguably more powerful than the tag syntax, although it is a little bit more verbose. So in this case, on line eight, I'm establishing a variable which is going to come from some other logic in my application. And then I've got the switch statement that standing all alone and immediately opening a curly brace.

```
func main() {
        i := 10
        switch {
        case i <= 10:
                fmt.Println("less than or equal to ten")
        case i <= 20:
                fmt.Println("less than or equal to twenty")
        default:
                fmt.Println("greater than twenty")
        }
}
```

- type switch

```
func main() {
        var i interface{} = 1
        switch i.(type) {
        case int:
                fmt.Println("i is an integer")
        case float64:
                fmt.Println("i is a float64")
        case string:
                fmt.Println("i is a string")
        default:
                fmt.Println(" anothr type")

        }
}
```

Defer

how we can actually invoke a function, but delay its execution to some future point in time.

```
func main() {
        fmt.Println("start")
        fmt.Println("middle")
        fmt.Println("end")

}
```

> In a normal go, application control flows from the top to the bottom of any function that we call.

do if we want to defer the execution one of these statements is proceeded with the defer keyword.

A defer statement pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. Defer is commonly used to simplify functions that perform various clean-up actions.

```
func main() {
        fmt.Println("start")
        defer fmt.Println("middle")
        fmt.Println("end")
}
//output
//start end middle
```

the way the defer keyword works in go, is that it actually executes any functions that are passed into it after the main function finishes its final statement, but before it actually returns. So the way this main function is executing is its calling start and it's printing out start, then it recognizes that it has a deferred function to call and then it prints out end. And then the main function exits now when go recognizes that that function exits, it looks to see if there are any deferred functions to call. And since we have one, and then goes ahead and calls that, so deferring doesn't move it to the end of the main function, and actually moves it after the main function. But before the main function returned.

```
func main() {
        defer fmt.Println("start")
        defer fmt.Println("middle")
        defer fmt.Println("end")
        // end middle start

}
```

> if we put the deferred keyword in front of all of these statements, then we'll actually see an interesting behavior. Because the deferred functions are actually executed in what's called LIFO order or last in first out.

> And this makes sense, because often we're going to use the deferred keyword to close out resources. And it makes sense that we close resources out in the opposite order that we open them, because one resource might actually be dependent on another one.

- For example, let's look at a function that opens two files and copies the contents of one file to the other:

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }

    written, err = io.Copy(dst, src)
    dst.Close()
    src.Close()
    return
}
```

This works, but there is a bug. If the call to os.Create fails, the function will return without closing the source file. This can be easily remedied by putting a call to src.Close before the second return statement, but if the function were more complex the problem might not be so easily noticed and resolved. By introducing defer statements we can ensure that the files are always closed

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}
```

another example

```
package main

import ("fmt"   ,"io/ioutil"    ,"log","net/http")

func main() {
        res, err := http.Get("http://www.google.com/robots.txt")
        if err != nil {
                log.Fatal(err)
        }

        defer res.Body.Close()
        robots, err := ioutil.ReadAll(res.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Printf("%s\n", robots)
}
```

a program that's going to make some resource requests through the HTTP package.
using the Get function from the HTTP package in order to request the robots dot txt file from google.com
we're going to get a response and an optional error. And we're going to check to see if that error is nil. And if it's not, then we're going to log that out and exit our application.
If err is nil, and we got a good response. So then we're going to use the read all function from the IO util package, what that'll do is that'll take in a stream, and that'll parse that out to a string of bytes for you to work with.
the defer keyword can help with is handling this Body.close.
using defer it allows you to associate the opening of a resource and the closing of the resource right next to each other.

*

```
func main() {
        a := "start"
        defer fmt.Println(a)
        a = "end"

}
```

> we're going to get the value "start" printed out. And the reason for that is when you defer a function like this, it actually takes the argument at the time, the defer is called not at the time the called function is executed.

panic

there are some things that get a go application into a situation where it cannot continue and that is considered exceptional. is going to be panic, because it can't figure out what to do.

```
func main() {
        fmt.Println("start")
        panic("some went Wrong!")
        fmt.Println("end")
}
```

- here we are trying to access a tcp port 8080 to serve our web app

```
package main

import (
        "net/http"
)

func main() {
        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
                w.Write([]byte("hello Go!"))
        })

        err := http.ListenAndServe(":8080", nil)
        if err != nil {
                panic(err.Error)
        }

}
```

when an err might happens, like when we try to access an already used port, so we measured it must panic to block this try
It's up to you, as a developer to decide whether that's a problem or not to panic over it, So what are we going to do in the situation that our application is panicking. And we can get ourselves to a situation where we can recover the application. So panics don't have to be fatal. They just are, if we panic all the way up to the go runtime, and the go runtime doesn't know what to do with a panicing application. And so it's going to kill it.

```
func main() {
        fmt.Println("start")
        defer fmt.Println("this was deferred")
        panic("something bad happend")
        fmt.Println("end")
}
```

> start
> this was deferred
> panic: something bad happend

we get this was deferred printing out and then the panic happens. And this is really important,
because panics happen after deferred statements are executed. So the order of execution is we're going to execute our main function, then we're going to execute any deferred statements, then we're going
to handle any panics that occur. And then we're going to handle the return value.

the first thing that's important is that the first statements that are going
to close resources are going to succeed even if the application panics.So if somewhere up the call stack,
you recover from the panic, you don't have to worry about resources being left out there and left open.
So any deferred calls to close resources are still going to work even if a function panics.

- recover from a panic

```
package main

import (
        "fmt"
        "log"
)

func main() {
        fmt.Println("starting...")
        panicker()
        fmt.Println("Ended")

}

func panicker() {
        fmt.Println("about to panic")
        // anonymous function
        defer func() {
                if err := recover(); err != nil {
                        log.Println("Error: ", err)
                }
        }() // these parenthesis are making that anon function to execute

        panic("something went wrong")
        fmt.Println("done panicking")
}
```

> output
> starting...
> about to panic
> 2021/11/22 11:43:29 Error: something went wrong
> Ended

> And an anonymous function is simply a function that doesn't have a name. So nothing else can call this, this is defined at one point, and we can call it exactly one time,

> an important thing to know about the defer statement, that it doesn't take a function itself, it actually takes a function call.

So what the recover function() is going to do is it will return nil if the application isn't panicking. But if it isn't nil, then it's going to return the error that actually is causing the application to panic.

we've got the main function, that's going to be our application entry point, of course. And then we've got this other function called panicker. And all this thing does is it's going to print on the line that says it's about the panic, and then it's going to panic. And it's going to go ahead and recover from that panic using that deferred function

we get the start line printed out, like you would expect, we see the about the panic string print out, then we panic, something bad happened, we go into our recover loop, because we're not going to execute this because our application panic. And so our panicker function is going to stop execution right there and execute any deferred functions. And inside of that deferred function, we call recover. So in that recover, we're going to log out the fact that we have that error, and we're going to log that error message out that we see here. But then in the main function execution continues.

the function that actually recovers, still stops execution, because it's in a state where it can no longer reliably continue to function. And so it makes sense for it to stop doing whatever it was trying to do. However, functions higher up the call stack, those functions that call the function that recovered from the panic, they are still presumably in a situation where they can continue, because you recover function said that your application is in a state working to continue to execute.

this is a little bit limiting as well, because in order to determine what that error is, you actually have to call the recover function, which basically means that you're saying that you're going to deal with it. So what happens if you get that error, and you realize that this isn't something that you can deal with? Well, in that case, what you're going to do is repanicing the application. So if you're in a situation where you're trying to recover from a panic, and you realize you can't handle it, you can feel free to re throw that panic, and the further management of that panic, higher up the call stack

Pointers

- creating pointers
- dereferencing pointers
- the new function
- working with nil
- types with internal pointers

a variable holds two different informations, the address where he is located, and the value in that location

> creating a pointer to reference a memeory address

```
func main() {
        var a int = 42
        var b *int = &a // declare pointer b to point to the address of a, so be is holding an address
        fmt.Println(&a, b) // here &a accesss its address in memory, b here already assigned that address, so both have same address

}
```

- before datatype means it declares a pointer to a memory address, like *int
- before variable means, access the value, like *b , it access the value

```
func main() {
        var a int = 42
        var b *int = &a // b is a pointer to an address
        fmt.Println(&a) // &a is access address of a

        fmt.Println(*b) // * is dereferencing the pointer to access the value of a

        //changing the value in the memory location, where a is located
        *b = 14
        fmt.Println(a, *b)

}
```

```
func main() {
        a := [3]int{1, 2, 3}
        b := &a[0] // b has the address of the first byte
        c := &a[1] // c has the address of the second byte
        fmt.Printf("%v %p %p\n", a, b, c)

}
// [1 2 3] 0xc000014150 0xc000014158
```

```
func main() {
        // ms is an object pointer that holds the address of myStruct ,
        // and it intialize the object with 52 value
        var ms *myStruct = &myStruct{foo: 52}
        fmt.Println(ms)
}

type myStruct struct {
        foo int
}
// &{52} this the address
```

- intialize a myStruct object with the new keyword

```
func main() {
        var ms *myStruct
        ms = new(myStruct)
        fmt.Println(ms)
}

type myStruct struct {
        foo int
}
// &{0}
```

we can't use the object initialization syntax, we're just going to be able to initialize an empty object. the zero value of a pointer is nil, the pointer that we don't intialize, it's going to be intialized to nil So this is very important to check in your applications. Because if you're accepting pointers as arguments, it is best practice to see if that pointer is a nil pointer. Because if it is, then you're going to have to handle that in a different way.

- how do we get to the underlying field of struct using a pointer using a dereferencing (*ms).foo is equal to ms.foo straight forward no need to * between parentheses

accessing the struct field to set its value after intializing myStruct using new `new()` in golang doesn't initialize memory but it only zeroes the value and return the pointer,In other words it returns a pointer to a newly allocated zero value of type T.

```
func main() {
        // new here create a m pointer of type map[string]int
        m := new(map[string]int)
        // intialize the map
        m = &map[string]int{
                "apple": 5,
        }
        v := *m
        v["cat"] = 1
        fmt.Println(v)
        fmt.Println(*m)
}
```

```
func main() {
        ms := new(myStruct) // new creates a pointer to myStruct
        ms.foo = 60 // this line equal to (*ms).foo=60
        fmt.Println(ms.foo)
}
type myStruct struct {
        foo int
}
// 60
```

how go handles variables when they're assigned one to another.

1. arrays to arrays as we know arrays make copies to the other variable

```
func main() {
        // a is an array
        a := [3]int{1, 2, 3}
        // b is pointing to same address as a, we must use pointer
        //because arrays copying the values not the address
        b := &a
        fmt.Println(a, b)
        a[1] = 8
        fmt.Println(a, b)
}
// [1 2 3] &[1 2 3]
// [1 8 3] &[1 8 3]
//  now any change in a affect b
```

2. slice are reference to the underlying array so slice to slice means both share the pointer to the same underlying array. no need to use pointer

```
func main() {
        // a is an array
        a := []int{1, 2, 3}
        // b is pointing to same address of slice a
        b := a
        fmt.Println(a, b)
        a[1] = 8
        fmt.Println(a, b)
}
//[1 2 3] &[1 2 3]
// [1 8 3] &[1 8 3]
```

using new keyword with slice to intialize a pointer s

```
package main

import (
        "fmt"
)

func main() {
        s := new([]string)
        fmt.Printf("length of string is %d and capacity is %d \n", len(*s), cap(*s))

        *s = append(*s, "test")
        //since length and capacity is 0, append function creates a new backing array and assigns to slice
        fmt.Printf("length of string after append is %d and capacity is %d\n", len(*s), cap(*s))

        *s = append(*s, "test1")
        fmt.Printf("length of string after append is %d and capacity is %d\n", len(*s), cap(*s))

        *s = append(*s, "test2")
        fmt.Printf("length of string after append is %d and capacity is %d\n", len(*s), cap(*s))

        fmt.Println(s)
}


output
length of string is 0 and capacity is 0
length of string after append is 1 and capacity is 1
length of string after append is 2 and capacity is 2
length of string after append is 3 and capacity is 4
&[test test1 test2]
```

3. maps to maps same as slice the referencing not copying

```
func main() {
        a := map[string]string{"foo": "bar", "baz": "buz"}
        b := a

        fmt.Println(a, b)
        a["foo"] = "woo"
        fmt.Println(a, b)
}
```

Functions

- basic syntax
- parameters
- return values
- anonymous functions
- functions as types ... how functions in go are first class citizens and can be passed around like any other variable
- methods
- passing parameters as value types passing in the variable by value. So that means that the go runtime is going to copy the data that's in this variable, since it copies, it shouldn't have affect on other copies when it changes

```
func main() {
        greeting := "hello"
        name := "hamo"
        sayGreeting(greeting, name)
        fmt.Println(greeting, name)

}

func sayGreeting(greeting, name string) {
        fmt.Println(greeting, name)
        name = "hoda"
        fmt.Println(greeting, name)
}


output
hello hamo
hello hoda
hello hamo
```

- passing parameters as pointer types instead of working with a copy of the variable, we're working with a pointer to the variable.
  we see that we have in fact, change the variable not only in the scope of the function, but in the calling scope as well. So by passing in a pointer, we have in fact manipulated that parameter that we passed in.

```
func main() {
        greeting := "hello"
        name := "hamo"
        sayGreeting(&greeting, &name)
        fmt.Println(greeting, name)

}

func sayGreeting(greeting, name *string) {
        fmt.Println(*greeting, *name)
        *name = "hoda"
        fmt.Println(*greeting, *name)
}


output
hello hamo
hello hoda
hello hoda
```

- why would you want to do this?

> a lot of times our functions do need to act on the parameters that are passed into them. And so passing in pointers is really the only way to do that. The other reason is passing in a pointer is often much, much more efficient than passing in a whole value. if you're passing in a large data structure, then passing in the value of that data structure is going to cause that entire data structure to be copied every single time.

- variadic paramters

```
func main() {
        sum(" The sum", 1, 3, 5, 8, 6)

}

func sum(msg string, values ...int) {
        fmt.Println(values)

        result := 0
        for _, v := range values {
                result += v
        }

        fmt.Println(result)
}


output
[1 3 5 8 6]
23
```

passing in the numbers one through five. Now, I'm not receiving five variables here, instead, I've got one variable here, and I've preceded its type with these three dots here. So what that's done is that's told the go runtime to take in all of the last arguments that are passed in, and wrap them up into a slice that has the name of the variable that we have.So since it's going to act like a slice, we can use a for loop and range over those values.

- using return type it's very useful to be able to use our functions to do some work, and then return a result back to the calling function.

```
func main() {
        s := sum(1, 3, 5, 8, 6)
        fmt.Println("the sum is", s)

}

func sum(values ...int) int {
        fmt.Println("values to be summed", values)

        result := 0
        for _, v := range values {
                result += v
        }

        return result
}
```

instead of printing the message in the sum function, we're returning the result out. in the main function, I can catch that return value by declaring a variable and setting it equal to the result of this function. it's just going to generate the result and return it back to the caller.

*_ return a local variable as a pointer_ So in our previous example, when we return that result, go actually copied that result to another variable, and that's what got assigned.

instead of returning the result, I'm returning the address of the result.

```
func main() {
        // s is now a pointer, becuase the return type of the sum is an address(pointer)
        s := sum(1, 3, 5, 8, 6)
        fmt.Println("the sum is", *s) // dereferenc s to get the values

}

func sum(values ...int) *int {
        fmt.Println("values to be summed", values)

        result := 0
        for _, v := range values {
                result += v
        }

        return &result
}
```

the ability to return a local variable as a pointer, like here result is a pointer but it is scope is inside the function
because when we declare the result variable, it's actually declared on the execution stack of this function, which is just a special section of memory that's set aside for all of the operations that this function is going to be working with. So in this func Exit, then execution stack is destroyed, that memory is freed up.

Go memory allocation - new objects, pointers and escape analysis

The pointer may escape to the heap, or it may not, depends on your use case. The compiler is pretty smart.

```
type Person struct {
    b, c int
}


func foo(b, c int) int {
    bob := &Person{b, c}
    return bob.b
}
```

It's all on the stack here, because even though bob is a pointer, it doesn't escape this function's scope.

However, if we consider a slight modification:

```
var globalBob *Person

func foo(b, c int) int {
    bob := &Person{b, c}
    globalBob = bob
    return bob.b
}
```

Then bob escapes to heap

- Named return Value

```
func main() {
        s := sum(1, 3, 5, 8, 6)
        fmt.Println("the sum is", s)

}

func sum(values ...int) (result int) {
        fmt.Println("the values to be summed", values)

        for _, v := range values {
                result += v
        }
        return
}
```

we don't have to do the maintenance of instantiating, this result variable

- Multipe Return values from function to demo this, see this example

```
func main() {
        d := divide(5.0, 3.0)
        fmt.Println(d)
}

func divide(a, b float64) float64 {
        return a / b
}
```

`output : 1.6666666666666667` as expect
But what happens if I pass in a zero here. Now when I run this, I get an unknown result, I get a positive infinity result. And I can't work with that in my application. So I'm going to probably cause some sort of a failure down the line. the only thing we could do is panic the application. But keep in mind when we talk about control flow and go, we don't want to panic our application as a general course of action, because panicking means the application cannot continue.

it's reasonable to assume that we might pass zero, what we actually want to do is return an error back letting the calling function know something that they asked it to do wasn'table to be done properly. we're actually going to add a second return variable. So to do that, we're going to add a print here. And we're going to return an object of type error

```
package main
import (        "errors"        "fmt"   "strings")

func capitalize(name string) (string, int, error) {
        handle := func(err error) (string, int, error) {
                return "", 0, err
        }
        if name == "" {
                return handle(errors.New("no name provided"))
        }
        return strings.ToTitle(name), len(name), nil
}
func main() {
        name, size, err := capitalize("sammy")
        if err != nil {
                fmt.Println("An error occurred:", err)
        }

        fmt.Printf("Capitalized name: %s, length: %d", name, size)
}
// output
//Capitalized name: SAMMY, length: 5
```

Here we use the 2 different return values from the call with multiple assignment. if you only want a subset of the returned values, use the blank identifier _

```
func vals() (int, int) {
        return 3, 7
}

func main() {
        a, b := vals()
        fmt.Println(a)
        fmt.Println(b)

        _, c := vals()
        fmt.Println(c)
}
```

```
package main
import ("fmt")

func divide(a, b float64) (float64, error) {

        if b == 0 {
                return 0.0, fmt.Errorf(" cannot divide by zero")
        }
        return a / b, nil
}
func main() {
        result, err := divide(3.0, 0.0)
        if err != nil {
                fmt.Println(err)
                return
        }
        fmt.Println(result)
}
```

- **anonymous functions**

. Functions are first-class citizens in Golang. What this means is that you can not only use function declarations as just reusable code blocks but you can also assign functions to variables, use functions as parameters on other functions, and even return functions from other functions. This is achieved using function literals which are also known as anonymous functions.

```
func main() {
        fmt.Println("Hello from main!")
        func() {
                fmt.Println("Hello from an anonymous function!")
        }()
}
```

1.Passing arguments You can pass arguments to these anonymous functions.

```
func main() {
        fmt.Println("Hello from main!")
        func(version float64) {
                fmt.Printf("Hello from an anonymous function in Go language %.2f!", version)
        }(1.15)
}
```

2. Assigning functions to variables

```
func main() {
        fmt.Println("Hello from main!")
        f := func(version float64) {
                fmt.Printf("Hello from an anonymous function in Go language %.2f!", version)
        }

        f(1.15)
}
```

3. Custom function types

> In GO, the function is also a type. Two functions will be of the same type if
> They have the same number of arguments with each argument is of the same type
> They have the same number of return values and each return value is of the same type

```
type concat func(fName, lName string) string

func main() {
        var s concat = func(fName, lName string) string {
                msg := fmt.Sprintf("%s %s rocks!", fName, lName)
                return msg
        }

        fmt.Println(s("Go", "Language"))
        fmt.Printf("%T", s)
}
```

```
type area func(int, int) int

func main() {
        var areaF area = func(a, b int) int {
                return a * b
        }

        // call inside a function as arg
        display(2, 3, areaF)

        // another call
        fmt.Println(" new Area is:", areaF(2, 5))
}

func display(x, y int, a area) {
        fmt.Printf("Area is: %d\n", a(x, y))
}
```

In this example also we create a user-defined function type area. Then we create a function getAreaFunc() which returns the function of type area

```
package main

import "fmt"

type area func(int, int) int

func main() {
        areaF := getAreaFunc()
        display(2, 3, areaF)

}

func display(x, y int, a area) {
        fmt.Printf("Area is: %d\n", a(x, y))
}

func getAreaFunc() area {
        return func(x, y int) int {
                return x * y
        }
}
```

4. methods A method is a function with a special receiver argument.
   In this example, the Abs method has a receiver of type Vertex named v

```
type Vertex struct {
        X, Y float64
}

func (v Vertex) Abs() float64 {
        return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
        v := Vertex{3, 4}
        fmt.Println(v.Abs())
}
```

You can declare a method on non-struct types, too.

```
type myFloat float64

func (f myFloat) Abs() float64 {
        if f < 0 {
                return float64(-f)
        }
        return float64(f)
}

func main() {
        f := myFloat(-math.Sqrt2)
        fmt.Println(f.Abs())
}
```

Pointer receivers

```
type Vertex struct {
        X, Y float64
}
func (v Vertex) Abs() float64 {
        return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func (v *Vertex) Scale(f float64) {
        v.X = v.X * f
        v.Y = v.Y * f
}
func main() {
        v := Vertex{3, 4}
        v.Scale(10)
        fmt.Println(v.Abs())
}


output
50
* scale 10: x=900, y=1600 because v is a pointer to the vertex it points to the new values
* sqrt 2500=50
```

Interfaces

- basics
- composing interfaces
- type conversion
    - the empty interfaces
    - type switches
- impementing with values vs pointers
- best practices

interfaces describe behaviors. storing method definitions

writer interface
the way this works is anything that implements this interface is going to take in that slice of bytes, write it to something that something might be the console, it might be a TCP connection, it might be the file system, we don't know, we just know that we're writing a slice of bytes to something. And then the integer and error that get returned. the error is there in case something goes wrong with the write operation. And the integer is normally the number of bytes written.

```
type Writer interface {
        // accept input slice of type byte, and return int and err
        write(data []byte) (int, error)
}
```

now that we have the interface defined, let's go ahead and implement it. So we're going to implement this with a console writer implementation, and that'll be a struct.

I've got a method on console writer called write. So it's got the same name as writer interface, it's accepting a slice of bytes, and it's returning an integer and an error. Now the implementation is whatever I want it to be. Now, in this case, all I'm going to do is convert that byte slice into a string and printed onto the console

```
type ConsoleWriter struct{}
func (cw ConsoleWriter) write(data []byte) (int, error) {
        n, err := fmt.Println(string(data))
        return n, err
}
```

the W variable is holding a writer, which is something that implements the writer interface.

```
        var w Writer = ConsoleWriter{}
        w.write([]byte("hello world"))
```

So when I call the write method, I know how to call that because that's defined by the interface. But I don't actually know in my main function, what's being written to, that's the responsibility of the actual implementation. So I could replace this with a TCP writer, I could replace it with a file writer, I could replace it with any other kind of writer. And so I get what's called a** polymorphic behavior**

- this the whole example

```
package main
import "fmt"
func main() {
        var w Writer = ConsoleWriter{}
        w.write([]byte("hello world"))
}
type Writer interface {
        // accept input slice of type byte, and return int and err
        write(data []byte) (int, error)
}
type ConsoleWriter struct{}
func (cw ConsoleWriter) write(data []byte) (int, error) {
        n, err := fmt.Println(string(data))
        return n, err
}
```

interface called incrementer. And that increment is going to be a method that only returns an integer, so it's going to increment something.
I defined the type alias for an integer called an Intcounter. And then a method to that custom type that's going to be my implementation for the incrementer interface. what I'm doing, I'm actually incrementing. The type itself, since I've got a type alias for an integer, it's a number, so I can go ahead and increment that. So I've actually got a type defined on an integer, and the integer itself is storing the data that the method is using.

```
func main() {
        myInt := IntCounter(3)

        var inc Incrementer = &myInt

        for i := 0; i < 6; i = i + 1 {
                fmt.Println(inc.Increment())
        }
}

type Incrementer interface {
        Increment() int
}

type IntCounter int

func (ic *IntCounter) Increment() int {
        *ic++
        return int(*ic)
}
```

intgers and their alias are passed by value, so to increment its identifier make a pointer if you want to keep modifying the new values

compose interfaces together

one of the keys to scalability

```
import (        "bytes" "fmt" )

func main() {
        var wc WriterCloser = NewBufferedWriterCloser()

        wc.Write([]byte("hello im hamo doing composite interfaces implementation "))
        wc.Close()
}

type Writer interface {
        Write(buf []byte) (int, error)
}

type Closer interface {
        Close() error
}

type WriterCloser interface {
        Writer
        Closer
}

type BufferedWriterCloser struct {
        buffer *bytes.Buffer
}
func (bwc *BufferedWriterCloser) Write(data []byte) (int, error) {
        n, err := bwc.buffer.Write(data)
        if err != nil {
                return 0, err
        }

        v := make([]byte, 8)
        for bwc.buffer.Len() > 8 {
                _, err := bwc.buffer.Read(v)
                if err != nil {
                        return 0, err
                }
                _, err = fmt.Println(string(v))
                if err != nil {
                        return 0, err
                }
        }
        return n, nil
}
func (bwc *BufferedWriterCloser) Close() error {
        for bwc.buffer.Len() > 0 {
                data := bwc.buffer.Next(8)
                _, err := fmt.Println(string(data))
                if err != nil {
                        return err
                }
        }
        return nil
}
func NewBufferedWriterCloser() *BufferedWriterCloser {
        return &BufferedWriterCloser{
                buffer: bytes.NewBuffer([]byte{}),
        }
}
```

```
package main

import ("fmt")

// Animal defines the interface for type Animal
type Animal interface {
        Says() string
        NumberOfLegs() int
}

// Dog defines the dog type
type Dog struct {
        Name  string
        Breed string
}

// Gorilla defines the gorilla type
type Gorilla struct {
        Name          string
        Color         string
        NumberOfTeeth int
}

func main() {
        // Create a dog variable of type Dog.
        dog := Dog{
                Name:  "Samson",
                Breed: "German Shepherd",
        }
        fmt.Println(fmt.Sprintf("The dog's name is %s and he is a %s.", dog.Name, dog.Breed))

        // Create a gorilla variable of type Gorilla.
        gorilla := Gorilla{
                Name:          "Geraldine",
                Color:         "black",
                NumberOfTeeth: 32,
        }
        fmt.Println(fmt.Sprintf("The gorilla's name is %s. She has %d teeth and she is %s in color.",
                gorilla.Name, gorilla.NumberOfTeeth, gorilla.Color))

        // We can pass dog to Riddle(), since the Dog type implements Animal interface by having all of the
        // necessary functions. The parameter is passed as a pointer since the functions for the type have pointer
        // receivers  and
        // We can also pass gorilla to Riddle(), since the Gorilla type satisfies the Animal interface.
        Riddle(&gorilla)
}

// Says has a receiver of type *Dog, so it satisfies part of the interface requirements for Animal
// for the Dog type
func (d *Dog) Says() string {
        return "woof"
}

// NumberOfLegs satisfies the rest of the Animal interface requirements for the Dog type
func (d *Dog) NumberOfLegs() int {
        return 4
}

// Says has a receiver of type *Gorilla, so it satisfies part of the interface requirements for Animal
// for the Gorilla type
func (g *Gorilla) Says() string {
        return "grunt"
}

// NumberOfLegs satisfies the rest of the Animal interface requirements for the Gorilla type
func (g *Gorilla) NumberOfLegs() int {
        return 2
}

// Riddle asks a riddle
func Riddle(a Animal) {
        info := fmt.Sprintf(`This animal says "%s" and
        has %d legs. What animal is this?`, a.Says(), a.NumberOfLegs())
        fmt.Println(info)
}
```