Go: The Complete Developer's Guide (Golang)

Created	@February 6, 2022 10:52 PM
Technology	GoLang
Status	Done
Platform	udemy
Type	

A simple start:

Hello world app:

```
package main

import "fmt"

func main() {
  fmt.Println("Hi There!")
}
```

1. How we do run the code inside the project?

   from terminal:

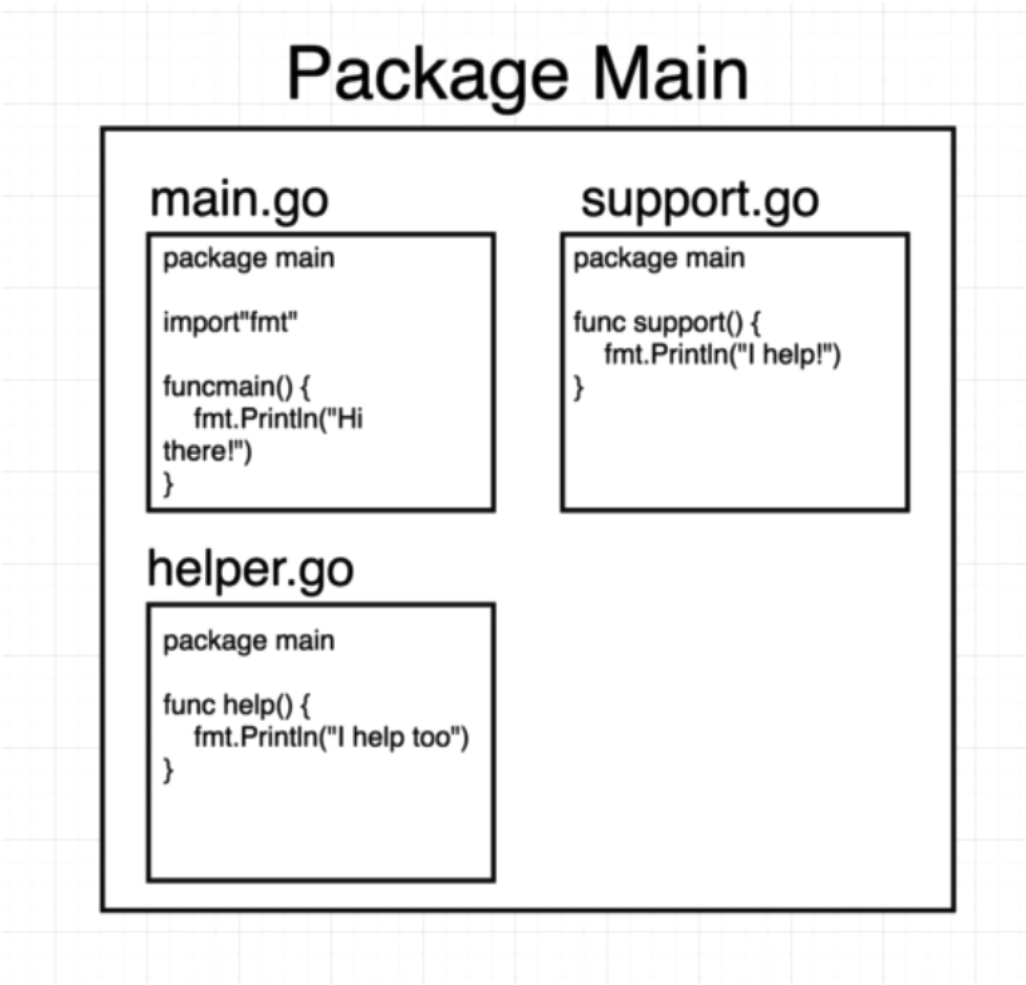
   `go run main.go`

   other commands:

   ```
   go build // compiles a bunch of go source code files

   go run // compiles and executes one or more two files

   go fmt // Formates all the code in each file in the current directory

   go install // Compiles and installs a package

   go get // Downloads the raw source code of someone else's package

   go test // Runs any tests associated with the current project
   ```

2. What does `package main` mean?

   package: is a project or workspace, collection of common source code files,
   in the first line of each file, we must declare the `package` that it belongs to. If we have multiple files that belong to `package main`, then it need to be defined in beginning of each one of them.



3. Why named the *package* main?

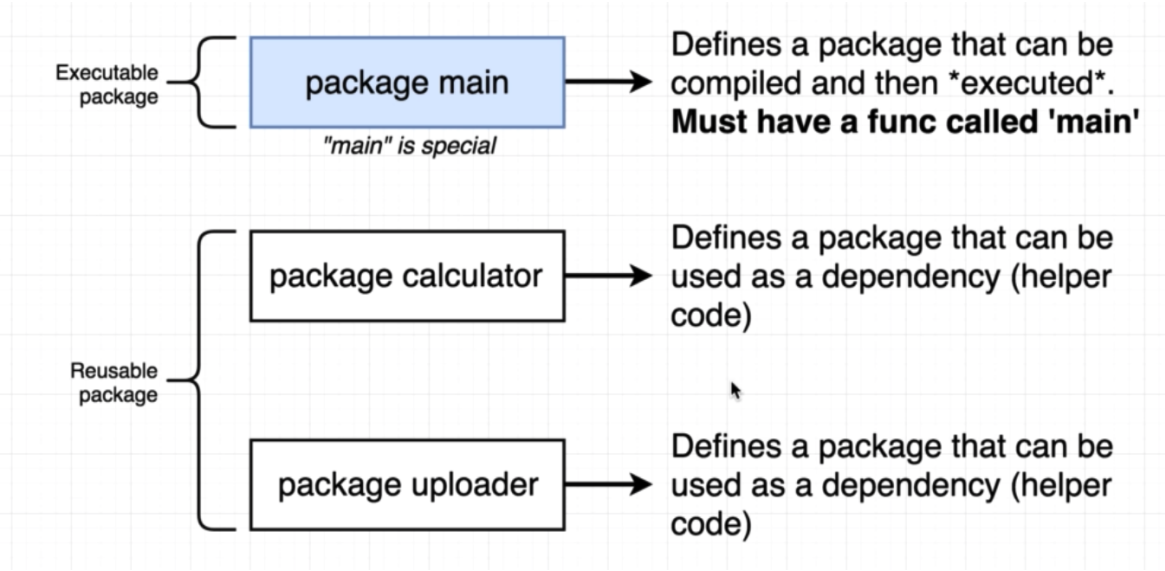
   Types of packages:

   Executable: Generates a file that we can run. and it does specific task.

   Reusable: Code used as 'helpers'. Good place to put reusable logic.

   The name of the `package` that you use that determines whether you are making an executable or dependency type package

   The word main defines the `package` as executable. If used any other name, It won't be executable, `main` packages must have a function inside it called `main`.



4. What does `import "fmt"` mean ?

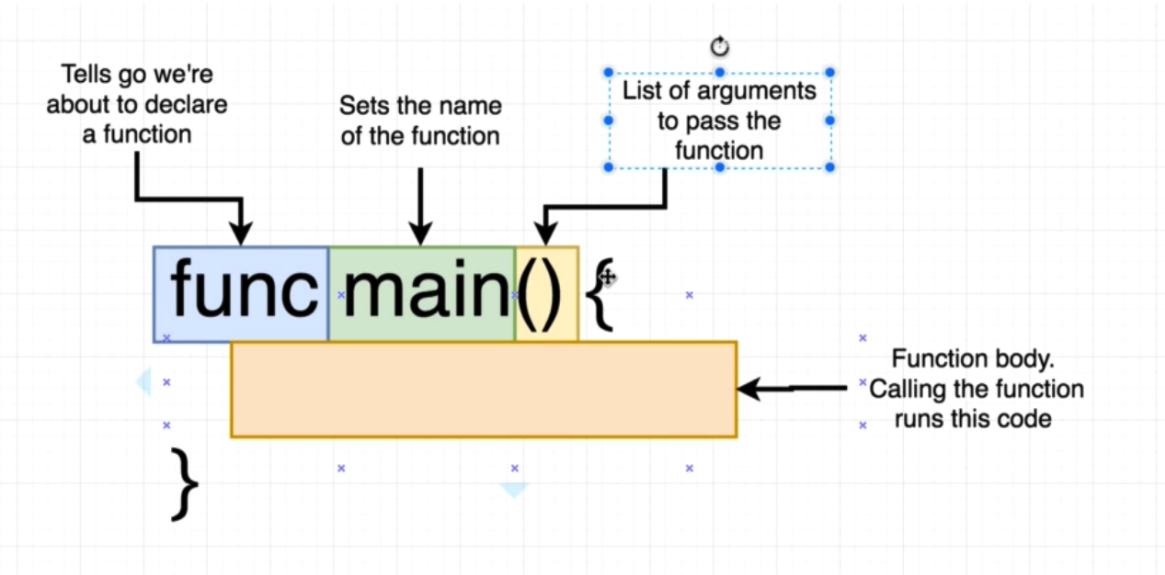
   Gives our package access to a code and functionality from another package called `fmt`

   `fmt`: is the name of a standard package that is included with the go programming language by default

   Package can have access to any number of packages by importing them.

5. What's that `func` thing?

   function



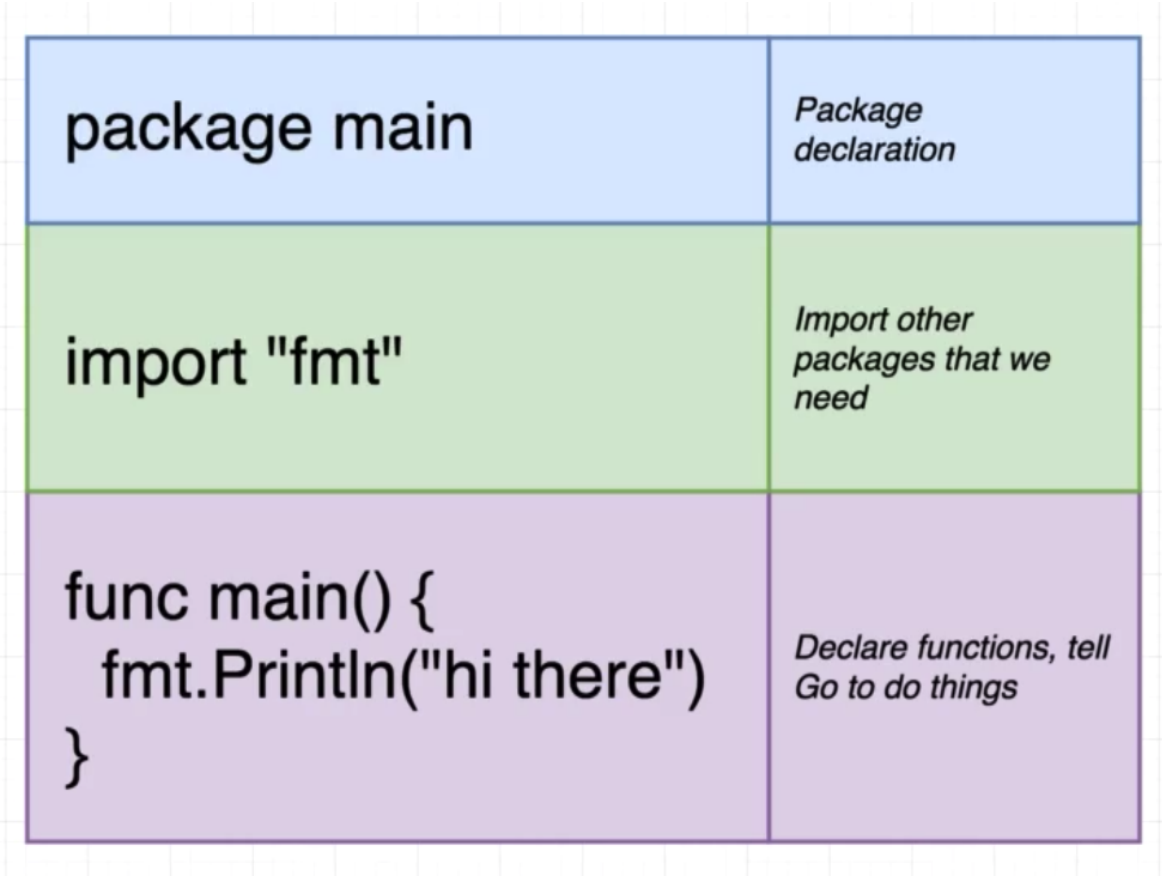
6. How is the `main.go` file organized?

   It has the same pattern:

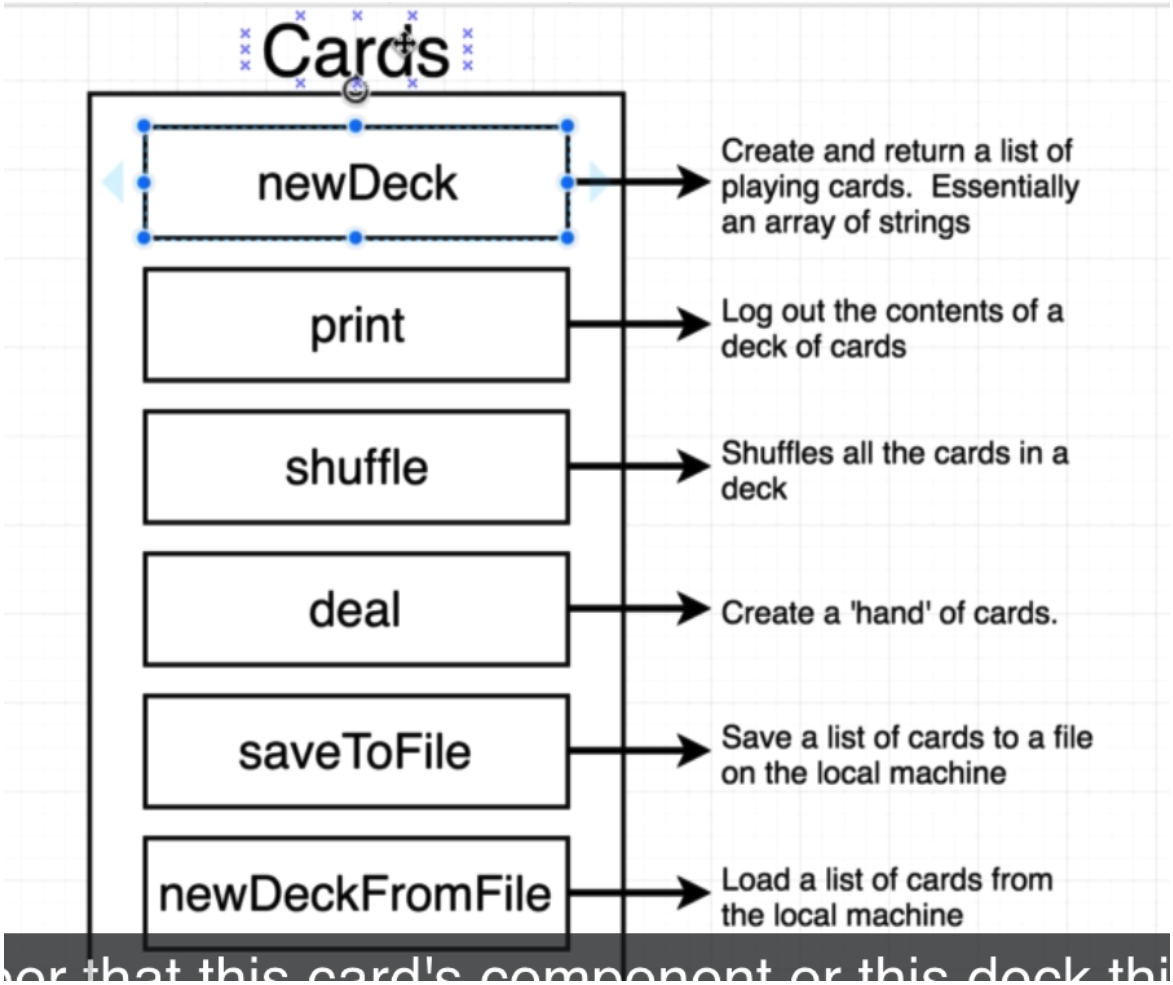
   In the top: `package` declaration

   after that: `import` other packages that we need

   then: declare functions, tell `go` to do things.



Deeper into Go:



1. **Defining a variable**: `var card string = "Ace of Spades"`
   `var`: we are about to create a new variable
   `string`: Only string will be ever assigned to this variable

   Go is a static type language, that's why we need to specify the type while defining the variable

   Common Types: `bool`, `string`, `int`, `float64`
   - `card := "Ace of Spades"` : this another way define a variable, in this version we are relying upon the go compiler to just figure out that variable is supposed to contain a string card type, It does that by reading in the colon equals operator, its better to use `:=`.
   - `card := newCard()` : We can call a function and insert to a variable

2. **Arrays and Slices**:

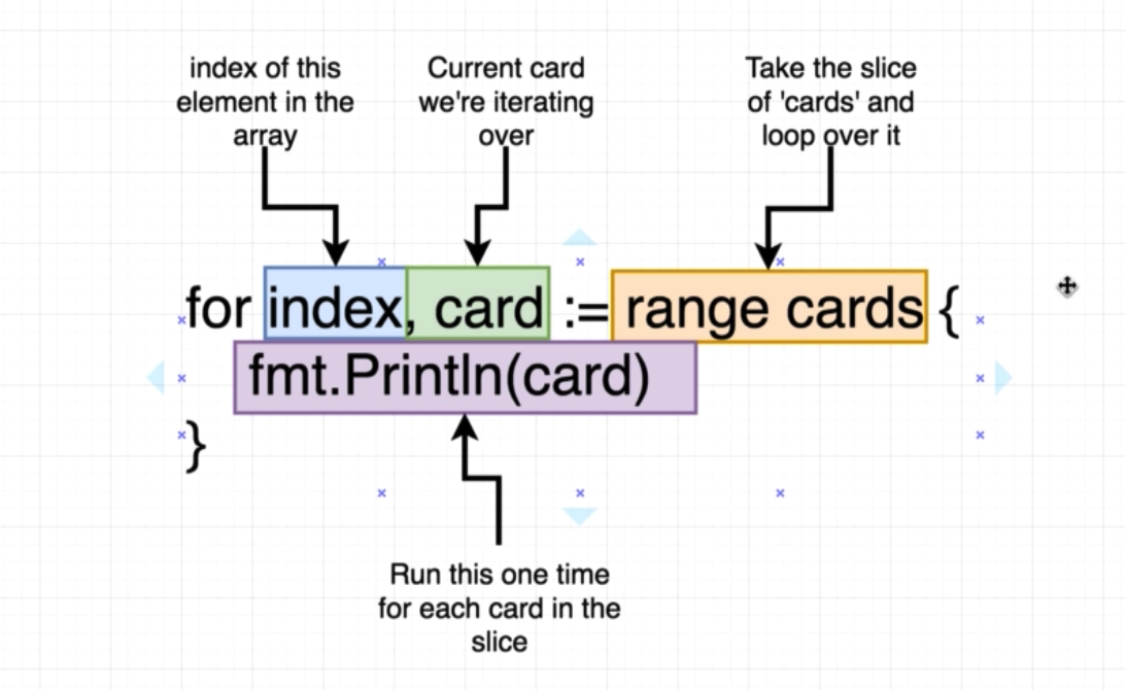
   Array: Fixed length list of things

   Slice: An Array that can grow and shrink

   Every element in Slices and Arrays must be of same type.

   - `cards := []string{"Ace of Diamonds", newCard()}` ⇒ This is how we declare slice
   - `cards = append(cards, "Six of Spades")` ⇒ This will not modify the original slice, instead it will return a new slice with added value
   - This is how we iterate over a slice, `range` is a keyword that we use whenever we want to iterate over every single record inside of a slice:

```
for i, card := range cards {
  fmt.Println(i, card)
}
```



3. **OO Approach vs Go Approach:**

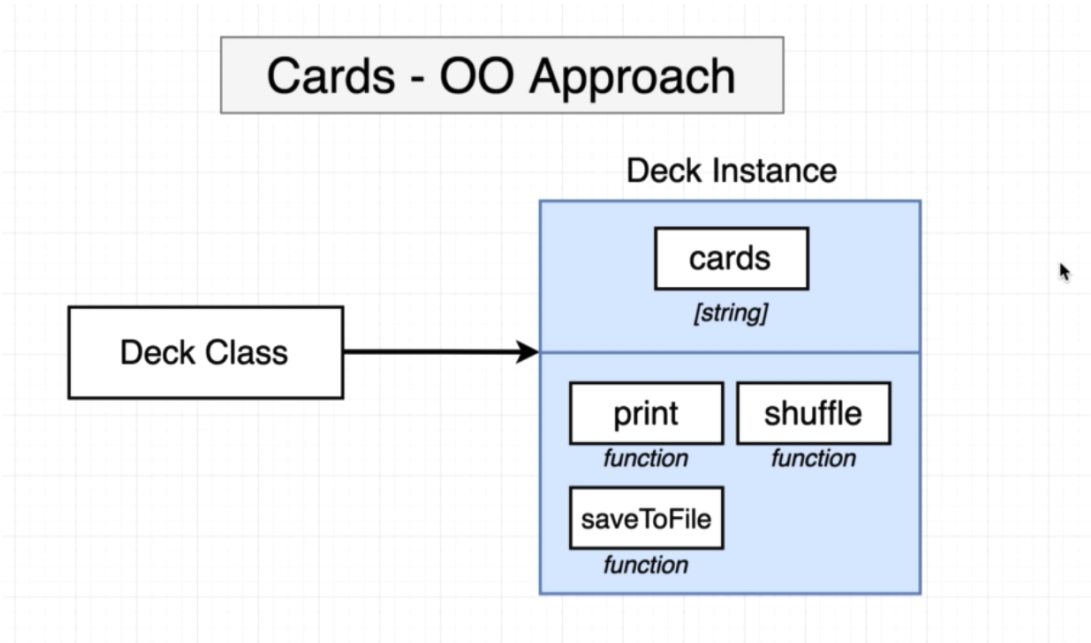
   Go is not an object oriented programming language, so there is no idea of classes inside of go.

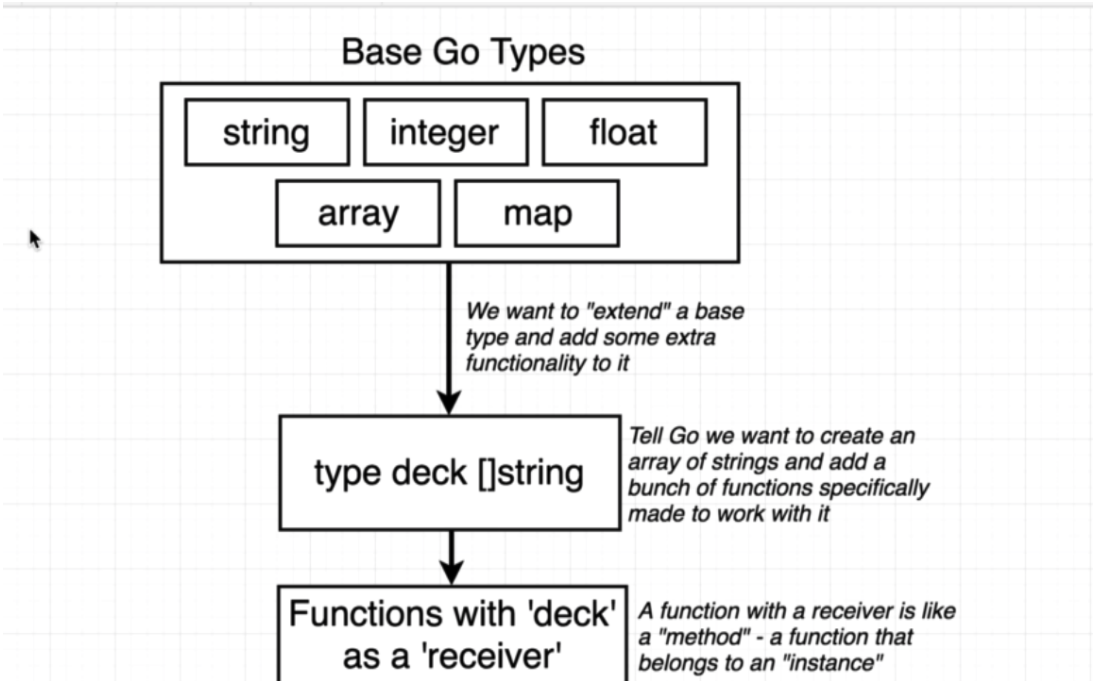
   In order to bring the idea of a `deck` (Project example) inside of our go program, We are going to define a new type inside of go

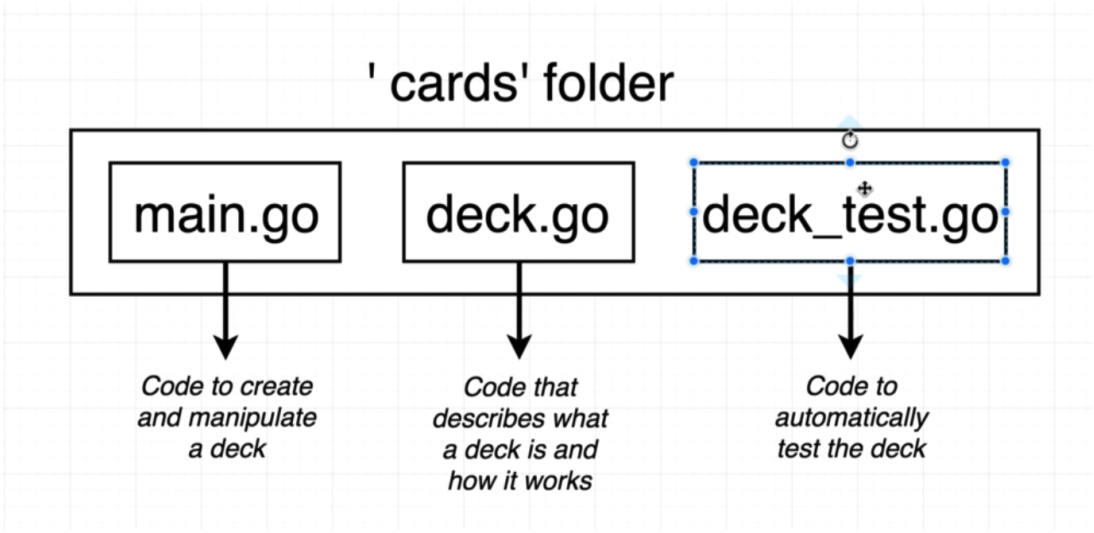
   We're going to define a new special type as being called a `deck` type, and a deck type will is going to be a slice of strings.

   The new type is essentially kind of a thin layer or a kind of abstraction of sorts over a slice of strings. So the new type behaves like a slice of string but we now have the ability to tie some additional functions specifically to anything that is of type `deck`.

   And in that slice of strings, we will have our list of playing cards, each of which will be a string







The deck type the same thing as a slice of string. But because we are making the extra type, it gives us the ability to create a bunch of custom functions. that only work with that deck type.

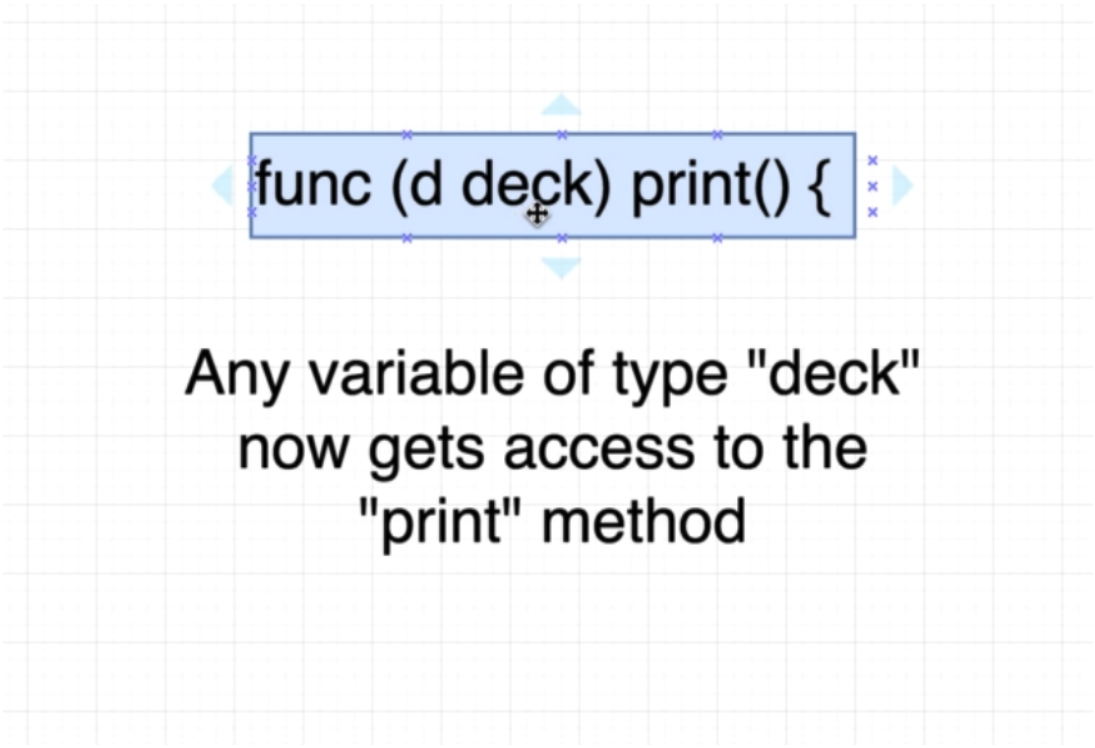
```
// To define a new type, you should define a new go package with the name that u want
// and inside it □□
package main

type deck []string // this is the same result as deck =: []string
```

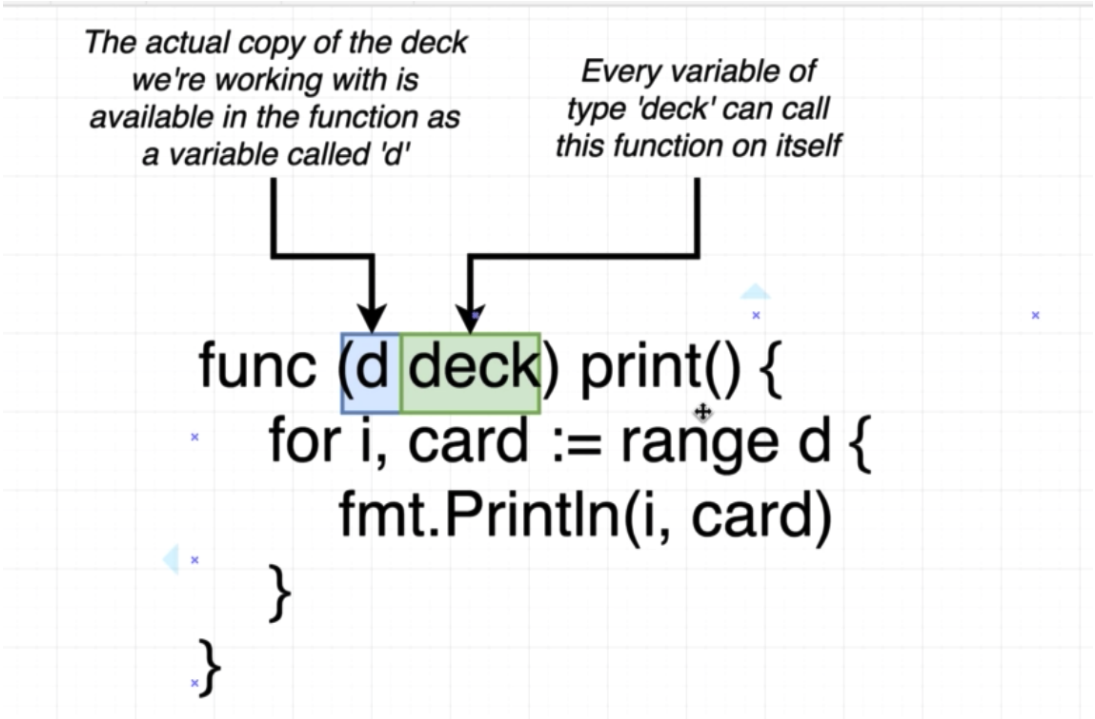
4. **Receiver:**
   In our example: we will add a function to the deck to print the slice, We will add a receiver to it and it will be of type deck, so any variable of type `deck` gets access to the `print` method

   `deck`: any variable of type deck will have access to the print method right before the word deck.

   `d`: receiver argument `d` is a reference to the actual working variable or the instance of the deck variable that we're working with. We choose `d` because by convention usually we refer to the receiver with a one or two letter abbreviation that matches the type that we of the receiver

```
cards.print() // d reperesents the cards variable
```





💡 **Note:** If you have an unused variable you can replace it with `_`

5. **Selecting elements from a slice:**

`fruits` ⇒ slice
`fruits[0]` ⇒ will select the first element
`fruits[0:2]` ⇒ select subset from the slice contain the first 2 elements, it include the first element from index range, and exclude the last element from the index range
`fruits[:2]` ⇒ same the above
`fruits[0:]` ⇒ means that it will return all elements from 0 index till the end of the slice

6. **Returning multiple values:**
   functions in Go can return multiple values:

```
func deal(d deck, handSize int) (deck, deck) { // This function will return 2 values of type deck
  return d[:handSize], d[handSize:]
}
```

   and to capture those returned values in variables, we need to do something like this:

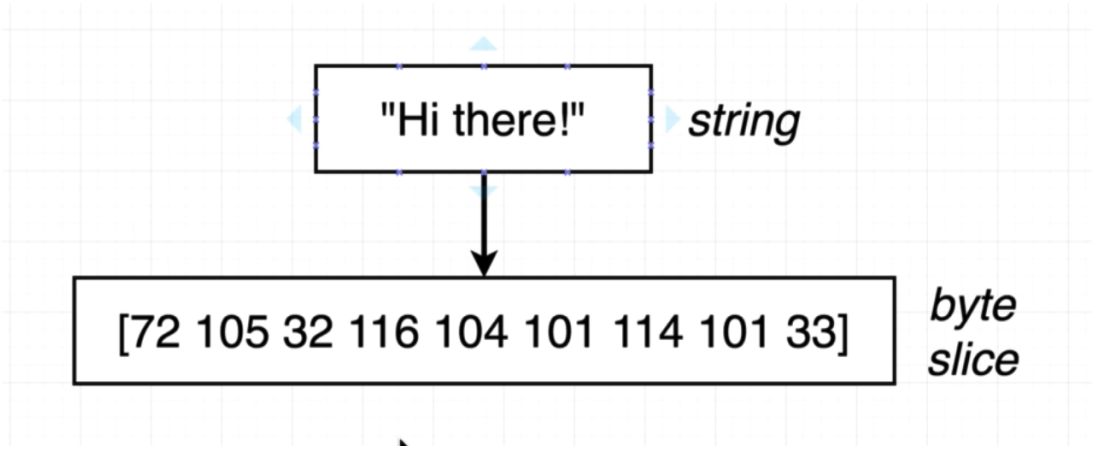
```
hand, remainingDeck := deal(cards, 5)
```

7. **Byte Slices:**

   We need to build a function to save to file. The intent of this function is to take a deck and somehow save that deck or a representation of that deck to the hard drive on our current machine. So we can later in the future be able to call new deck from file, which will load up that file from the hard drive and create a new deck or a new list of cards. And here we will use Go's standard library packages. The package is called IO Util. But first we need to convert our data to Byte Slice so we can pass it to the package.

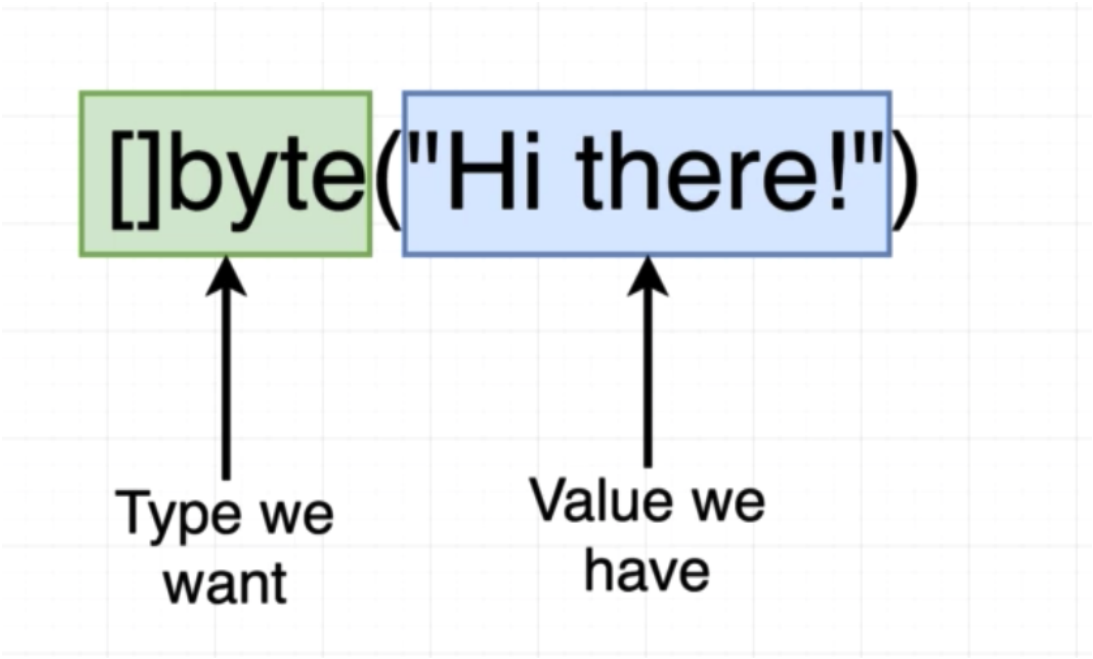
   Byte Slice: Slice like an array where every element inside of it corresponds to an ASCII character code.

   So if we represented this with decimals, we would have for the string "Hi there!", the below slice👇 :



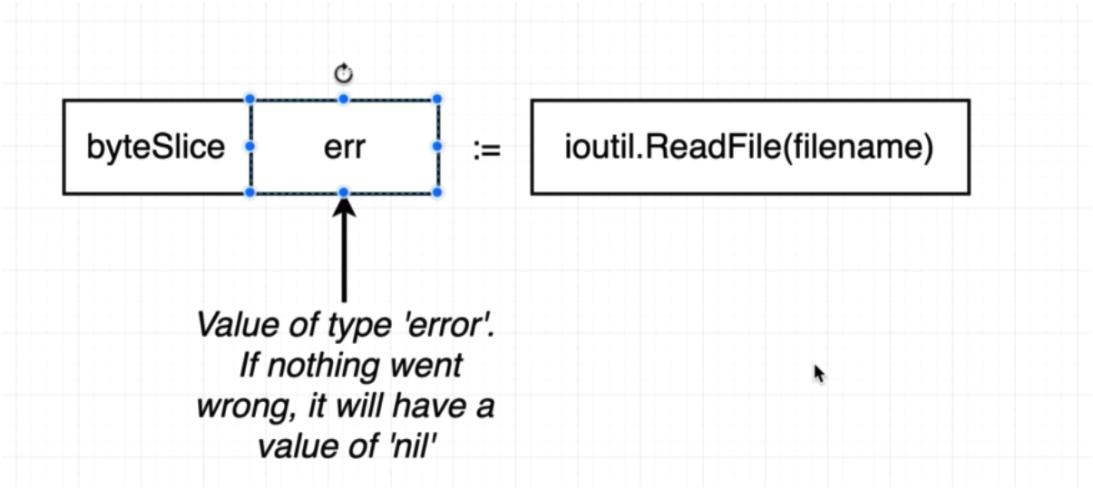
8. **Deck Type to String:**

   Type conversion with Go which is used to take one type of value and turn it into another type.



9. **ReadFile:**

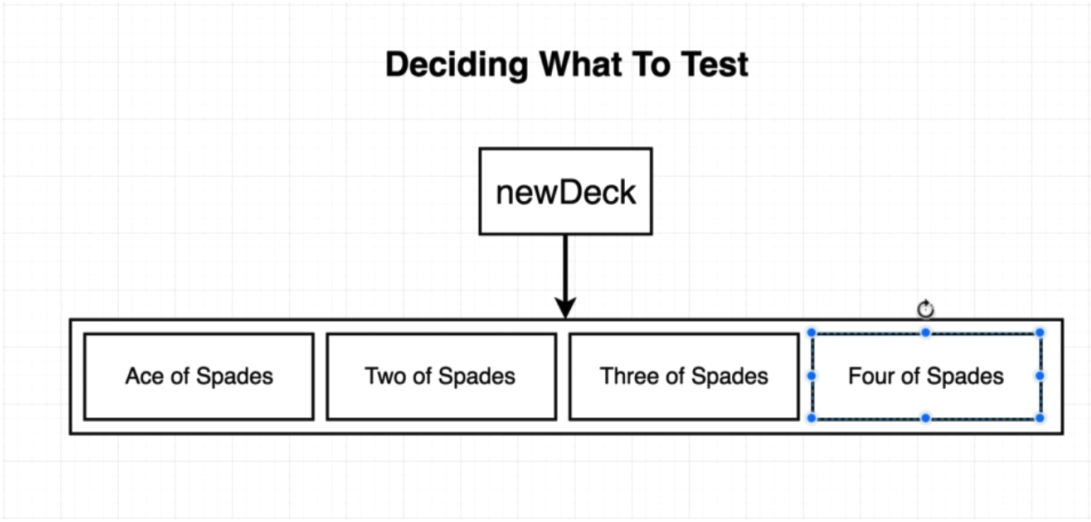
   ReadFile may return errors sometimes, so this how error is handled:

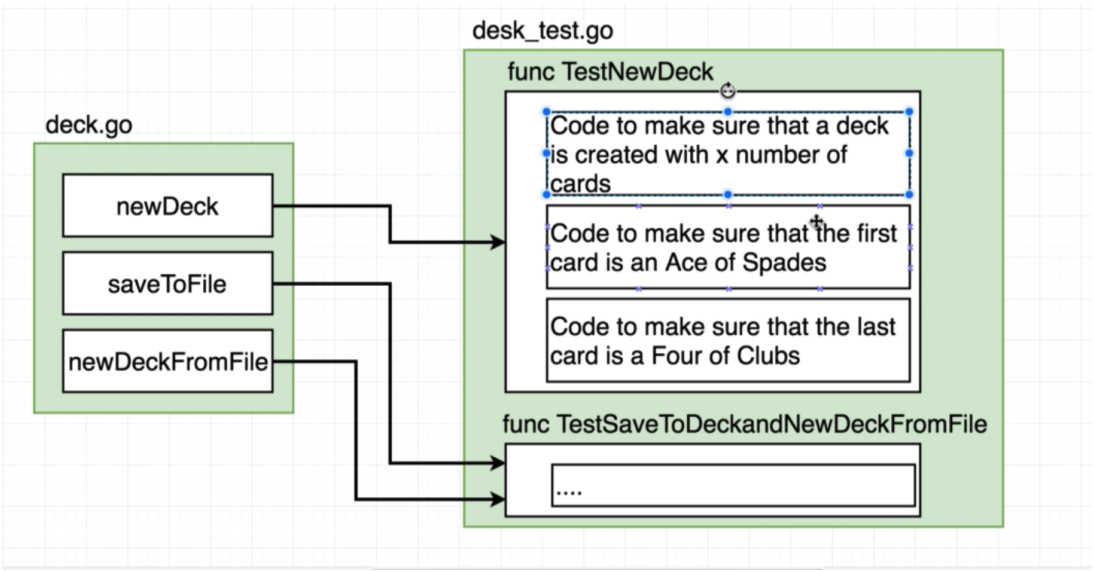


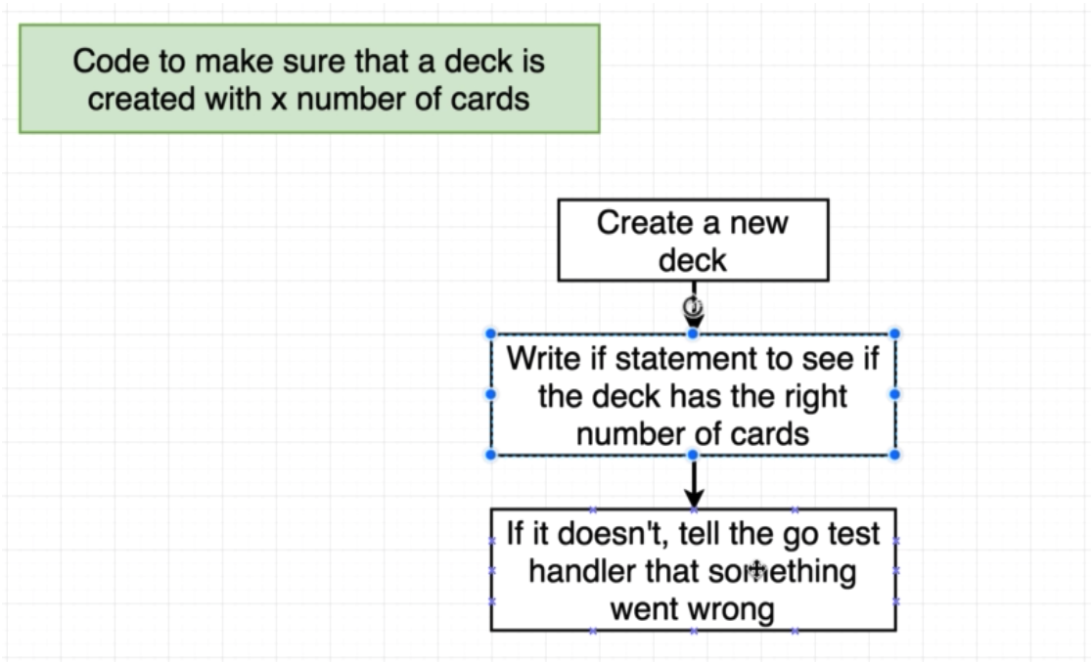
10. **Testing with Go:**

    Go testing is not RSpec, mocha, jasmine, selenium, etc!
    To make a test, create a new file ending in `_test.go`
    And to run all tests in a package, run the command: `go test`

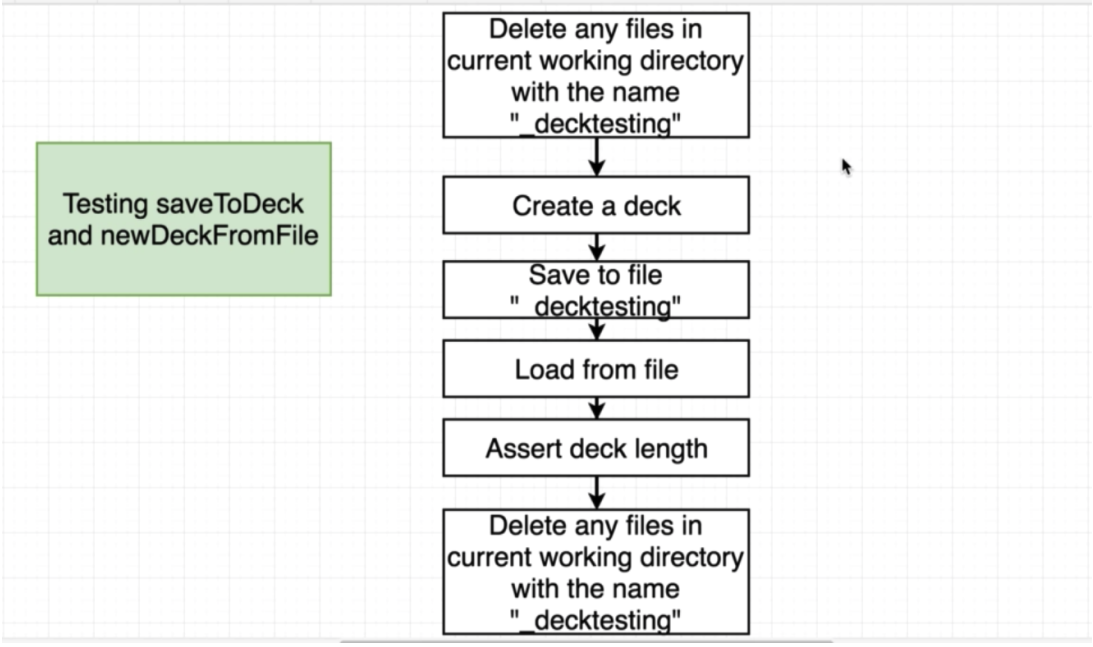
    The test file should contain a Go code that run against the target package to test it.
    The func inside this file should start with `Test` , ex: `TestNewDeck()`


Deciding What To Test
newDeck
Ace of Spades
Two of Spades
Three of Spades
Four of Spades


desk_test.go
func TestNewDeck
Code to make sure that a deck is created with x number of cards
Code to make sure that the first card is an Ace of Spades
Code to make sure that the last card is a Four of Clubs
deck.go
newDeck
saveToFile
newDeckFromFile
func TestSaveToDeckandNewDeckFromFile
....

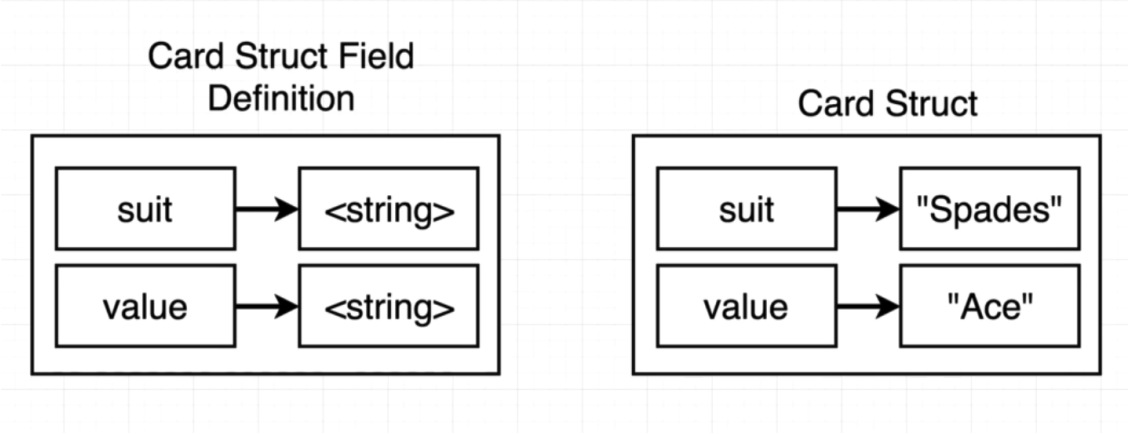

Code to make sure that a deck is created with x number of cards
Create a new deck
Write if statement to see if the deck has the right number of cards
If it doesn't, tell the go test handler that something went wrong

The cleanup in the testing need to done manually, so you need to clean it up by yourself.



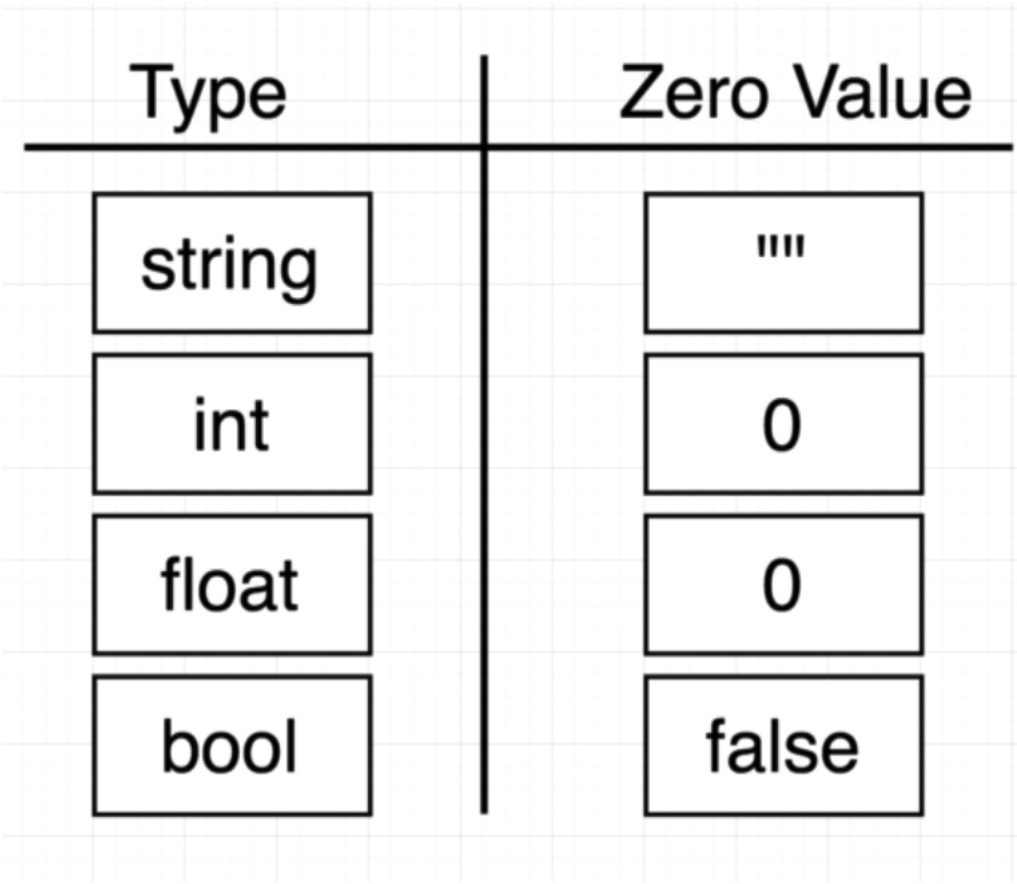
Organizing data with structs:

Data structure. Collection of properties that are related together.




```
// Definition of struct of type person
type person struct {
  firstName string
  lastName  string
}

func main() {

  alex := person{firstName: "Alex", lastName: "Anderson"} // initialize values for the person struct
  fmt.Println(alex)
}
```

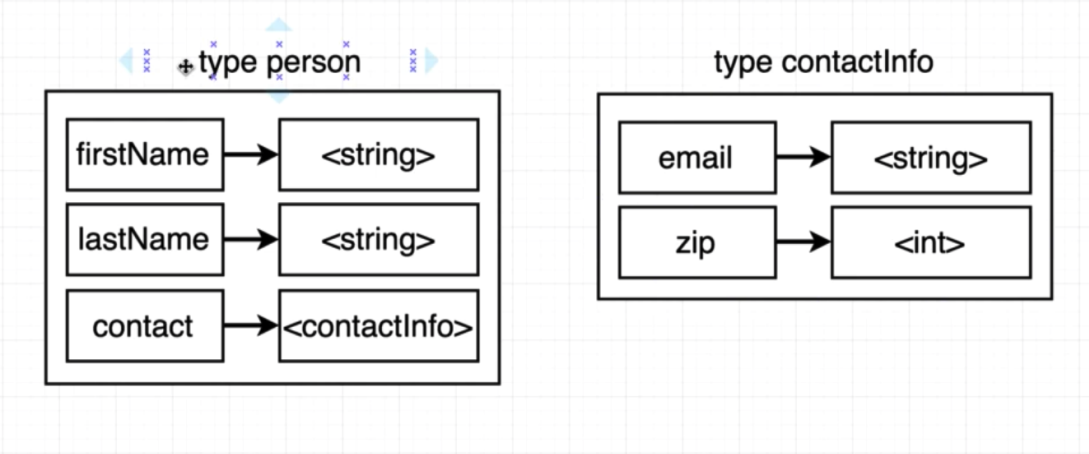
1. **Updating struct values:**
   When you define a variable and didn't assign a value to it, Go assigns what is referred to as a zero value of these different fields inside the struct.



You can us `fmt.Printf("%+v", alex)` to print a struct with its contents

```
// Definition of struct of type person
type person struct {
  firstName string
  lastName  string
}

func main(){
  var alex person
// This is how to update struct values
  alex.firstName = "Alex"
  alex.lastName = "Anderson"

  fmt.Println(alex)
  fmt.Printf("%+v", alex)
}
```

2. **Embed Struct:**
   This is used to embed a struct inside another struct, for example: we want to embed contact info inside person struct as contact.




```
package main

import "fmt"

type contactInfo struct{
  email string
  zipCode int
}

// Definition of struct of type person
type person struct {
  firstName string
  lastName  string
  contact contactInfo // define that contact is contactInfo type
}

func main() {
  jim := person{
    firstName: "Jim",
    lastName: "Party",
    contact: contactInfo{
      email: "jim@gmail.com",
      zipCode: 9400,
    }
  }

  fmt.Printf("%+v", jim)
}
```

Structs with Receiver functions:

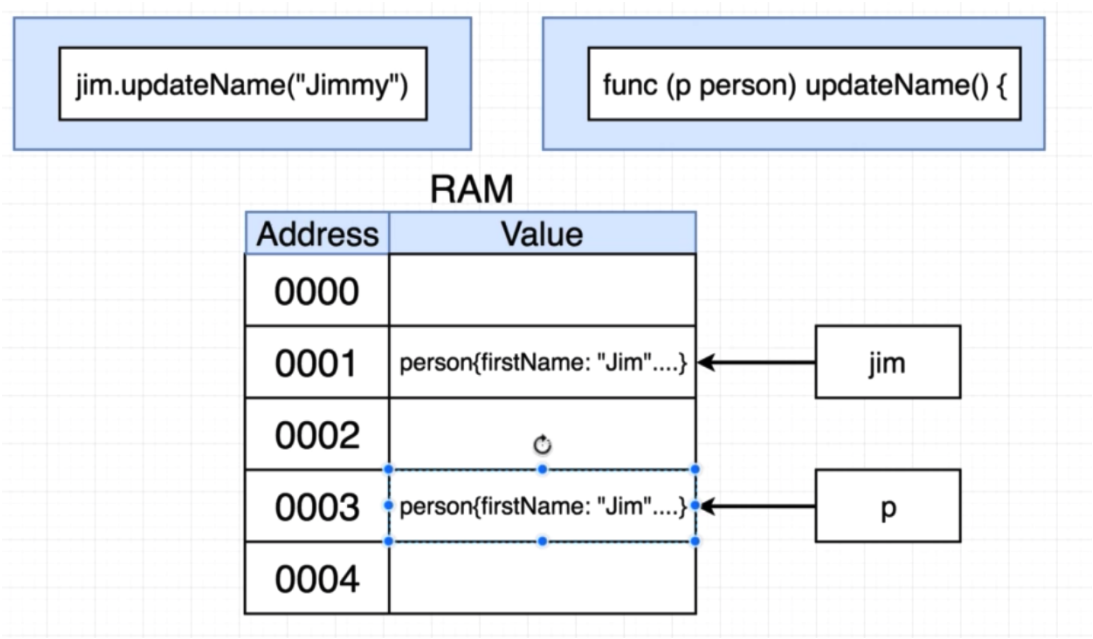
There is another way to embed struct into another struct, This can be done by just writing the struct type instead of defining a key to insert the struct into it.

```
package main

import "fmt"

type contactInfo struct{
  email string
  zipCode int
}

// Definition of struct of type person
type person struct {
  firstName string
  lastName  string
  contactInfo // define that embed struct is contactInfo type
}

func main() {
  jim := person{
    firstName: "Jim",
    lastName: "Party",
    contactInfo: contactInfo{
      email: "jim@gmail.com",
      zipCode: 9400,
    }
  }
```

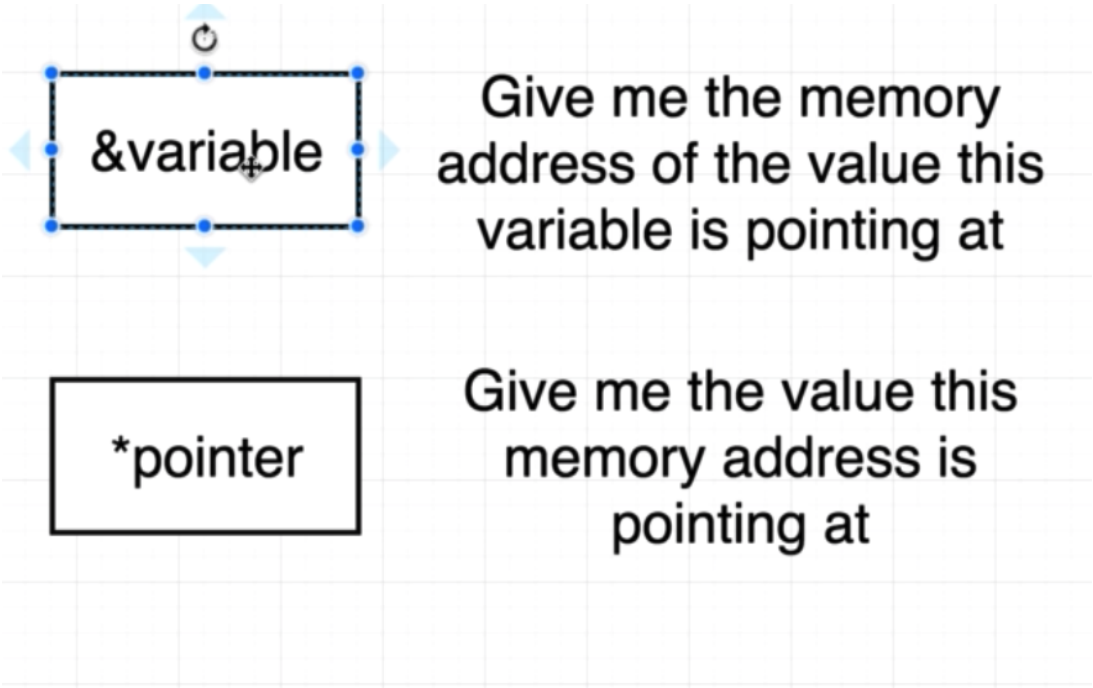
```
  jim.print()
}

func (p person) print() { // defining a receiver function for the struct
  fmt.Printf("%+v", p)
}
```

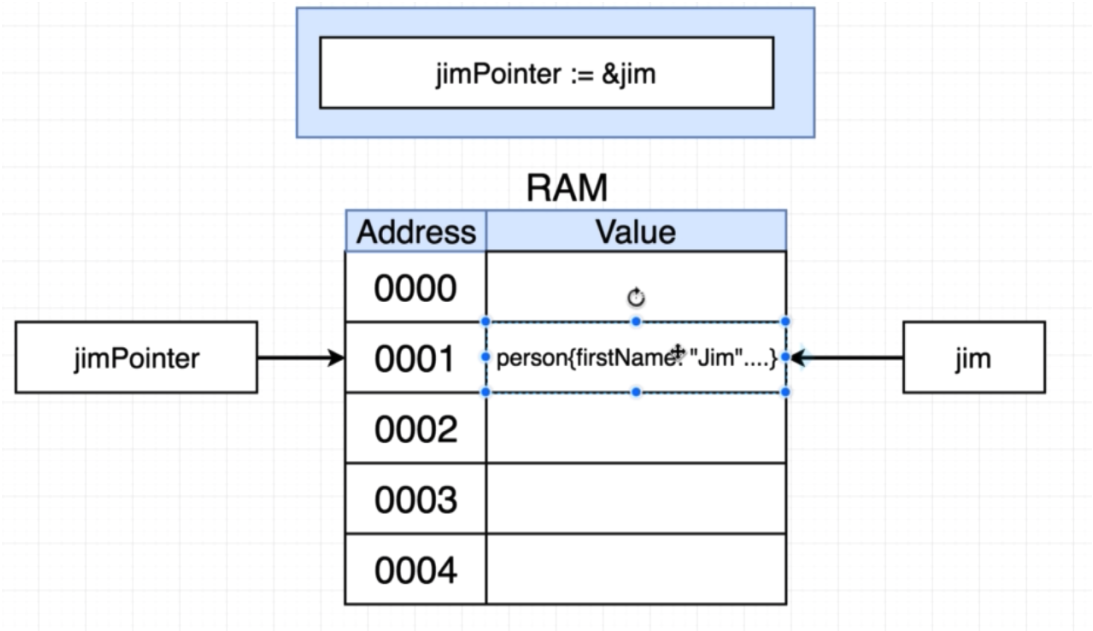
3. **Pass by value:**

   Go is a pass by value language, means whenever we pass some value into. a function Go will take that value or that struct and It's going to copy all of that data inside that struct and then place it inside of new some new container. When we pass a struct ti a func, Go make a copy of the entire struct.



Pointer Operations:

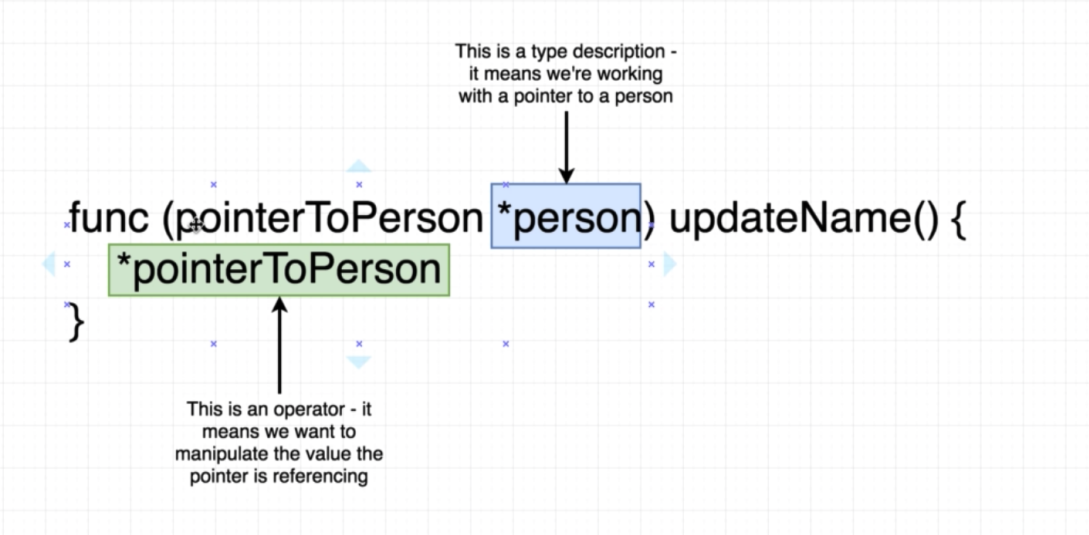


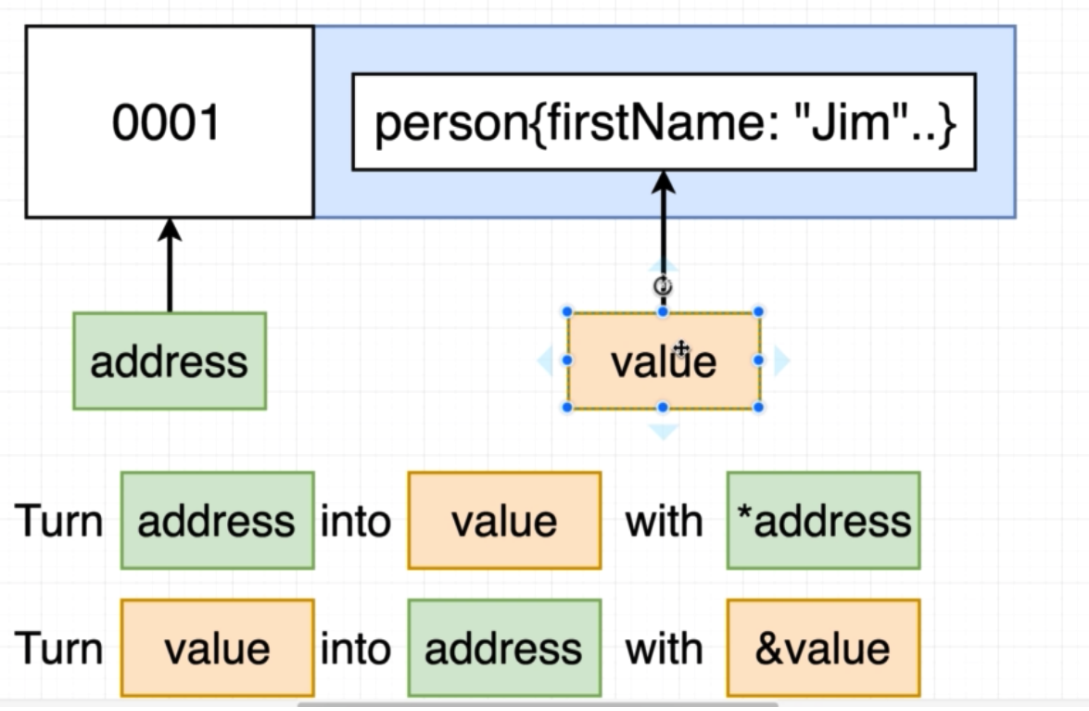


When we use a `*` in front of a type, it means something completely different than when we see a star in front of an actual pointer.

In front of a type: This is a type description, It means we are working with pointer to a person, means that this update function can only be called with the receiver of a pointer to a person.

Inside the function: this is an actual operator that takes the pointer and converts it into the actual value.



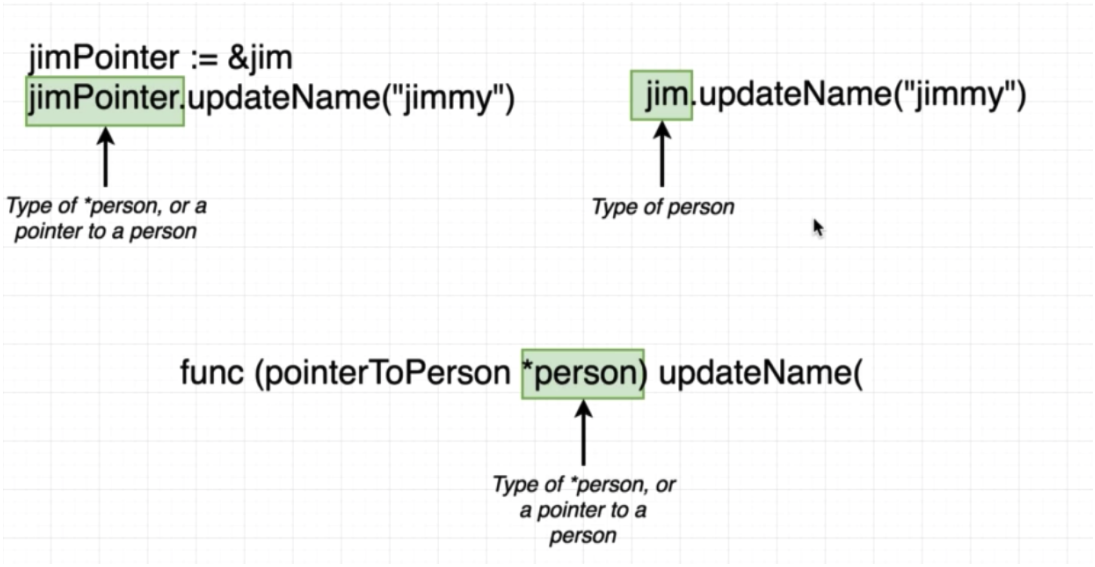


Pointer Shortcut:

In Go, If you define a receiver with a type of a pointer to whatever like pointer of blank, any type when we attempt to call this function, Go will allow us to either call this function with a pointer or with a core type person like this example:

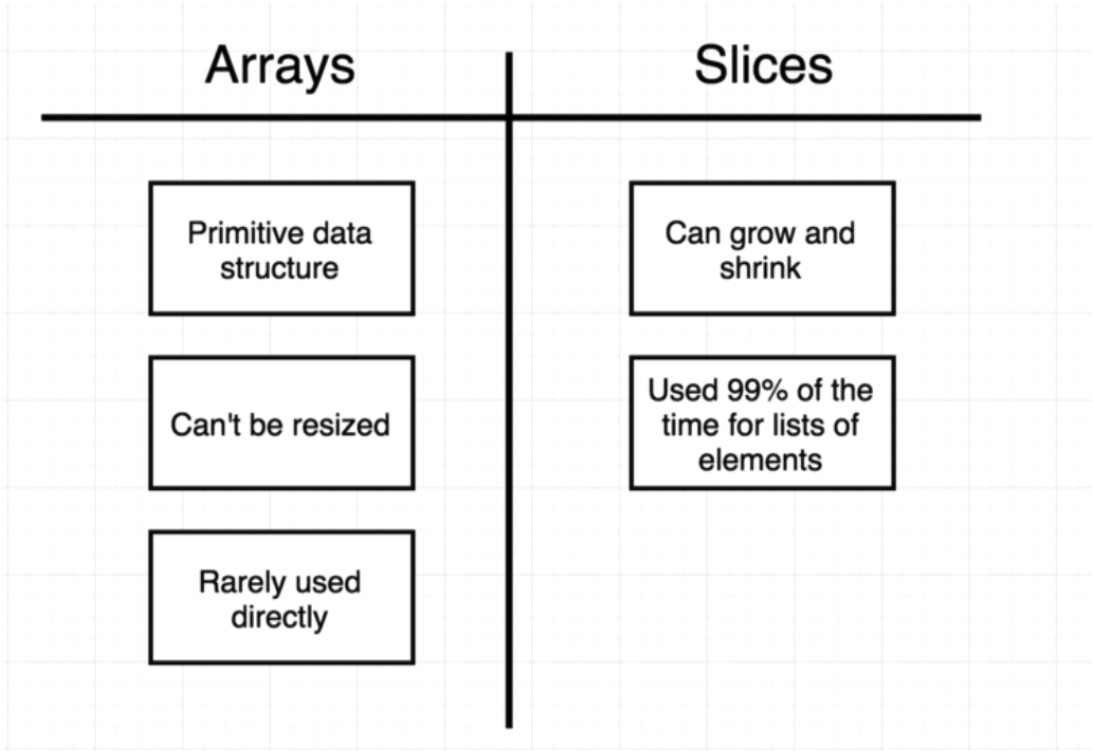
```
package main

import "fmt"

type contactInfo struct {
  email   string
  zipCode int
}

// Definition of struct of type person
type person struct {
  firstName string
  lastName  string
  contactInfo
}
```

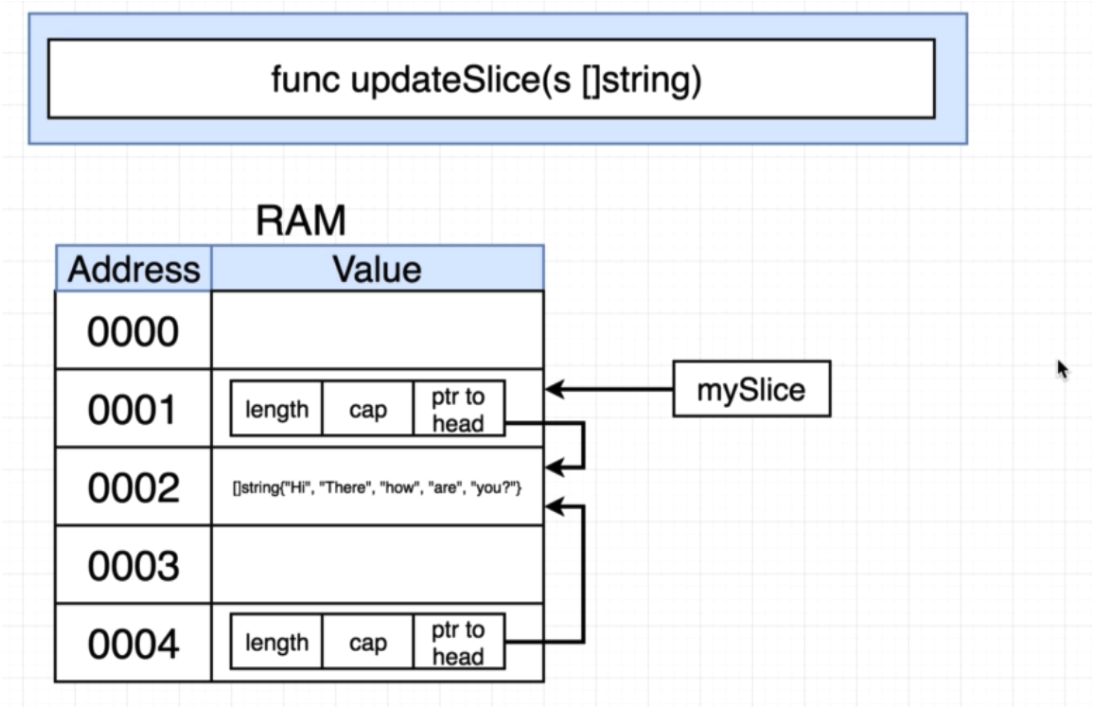
```
func main() {
  jim := person{
    firstName: "Jim",
    lastName:  "Party",
    contactInfo: contactInfo{
      email:   "jim@gmail.com",
      zipCode: 9400,
    },
  }

  jim.updateName("jimmy") // We called th e function through the root person type
  jim.print()
}

func (pointerToPerson *person) updateName(newFirstName string) {
  (*pointerToPerson).firstName = newFirstName // Whenever we use * before a pointer, this turns the pointer into value
}

func (p person) print() {
  fmt.Printf("%+v", p)
}
```



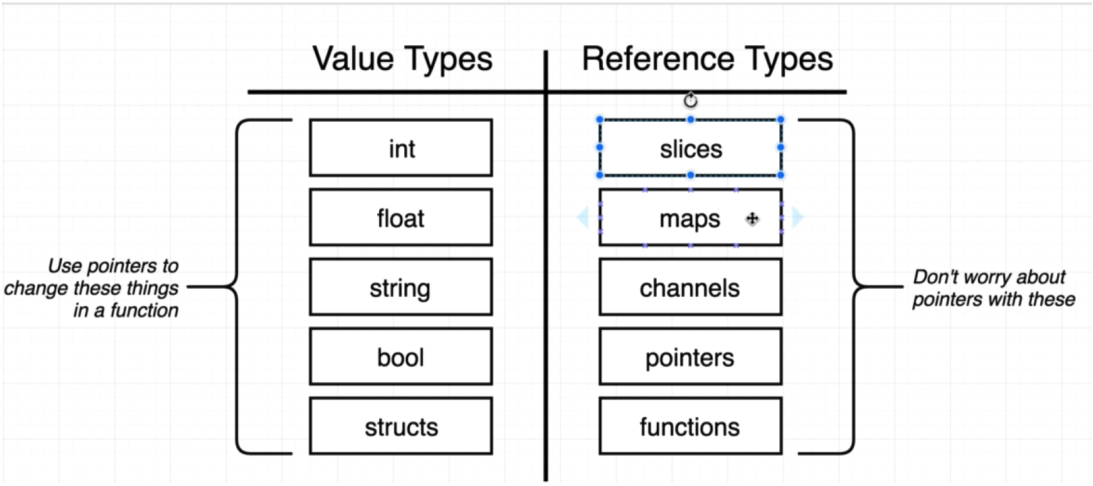
4. **Reference vs. Value types:**

   In slice and Array, the case is different, they are passed by reference, so whenever you pass a slice as a function and modify it, this modify the original slice.





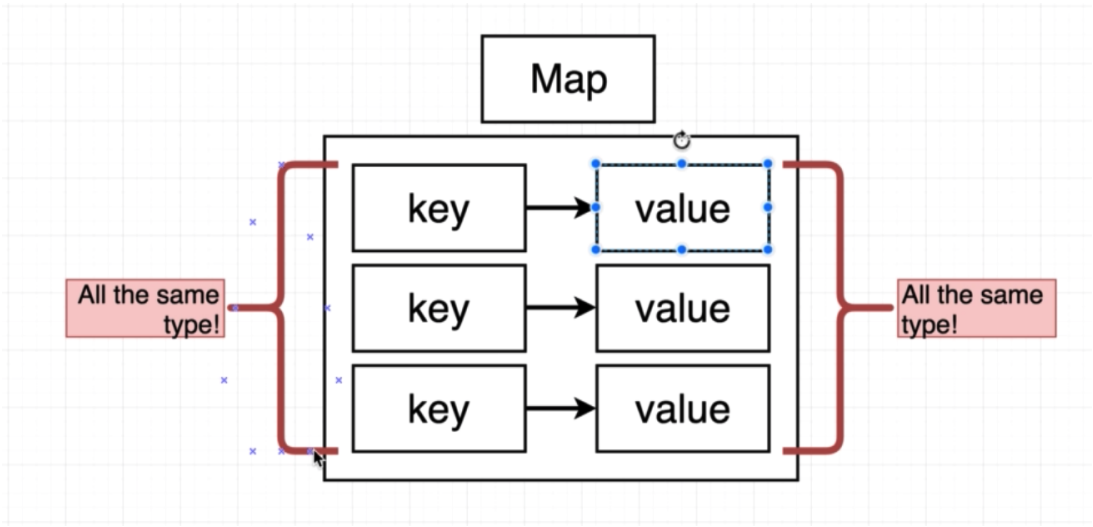
Slices are not the only reference type.



Maps:

Maps is very similar to structs. A Map is a collection of key-value pairs (Hash in Ruby)

Both keys and values are statically typed, So whenever we add some number of keys to a map, they all must be of the same exact type, And all the different values that we add as well must also be of the exact same type. The keys and the values themselves don't have to be of the same type, just all the different values have to.

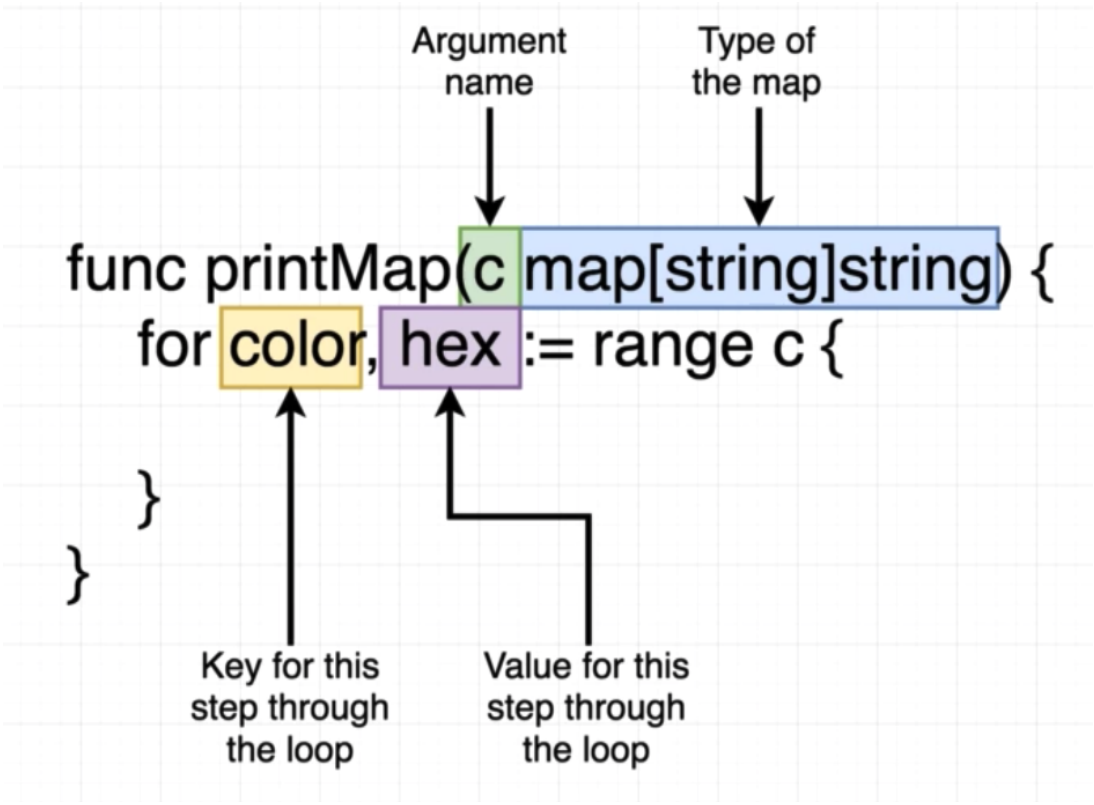



```
package main

import "fmt"

func main() {
  colors := map[string]string{ // This is how to declare a map
    "red":   "#ff0000",
    "green": "#4b5f5r",
  }

  fmt.Println(colors)
}
```

```
package main

import "fmt"

func main() {
```

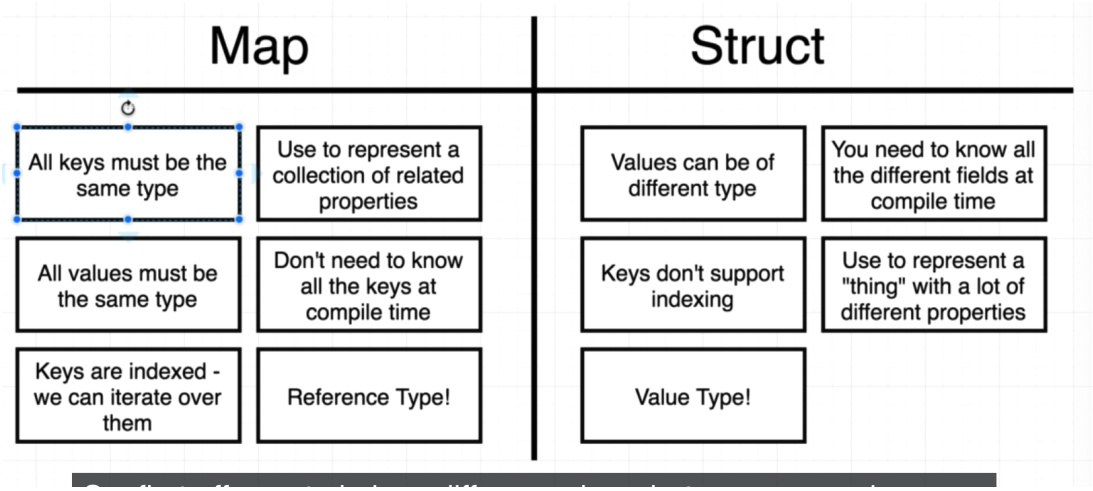
```
  // var colors map[string]string

  colors := make(map[string]string) // This is aother way to define a map

  colors["white"] = "#ffffff"

  delete(colors, "white") // To delete a value from a map
  fmt.Println(colors)
}
```

Iterating over a map:




```
func printMap(c map[string]string) {
  for color, hex := range c {
    fmt.Println("Hex code for", color, "is", hex)
  }
}
```

Map vs Structs:

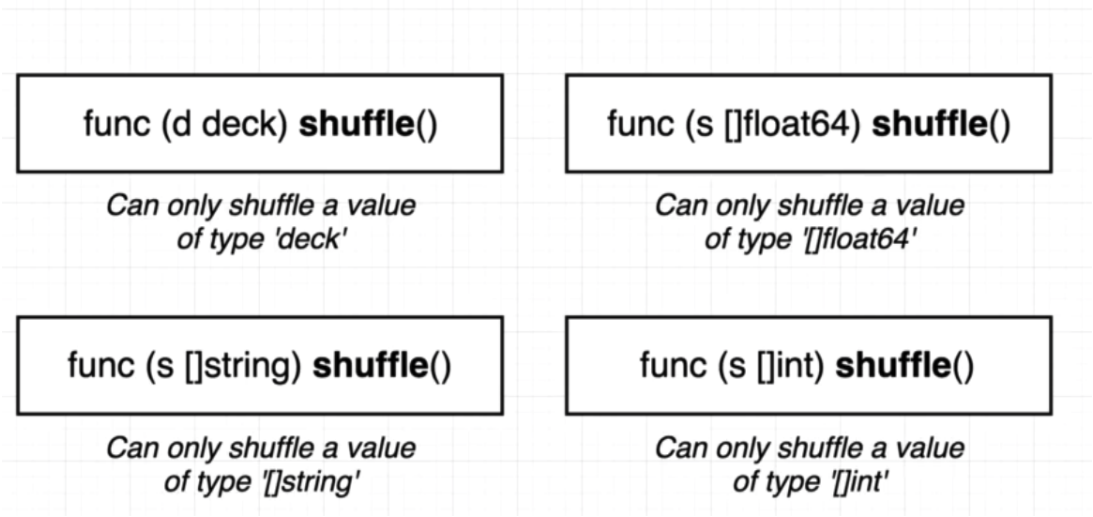


Interfaces:

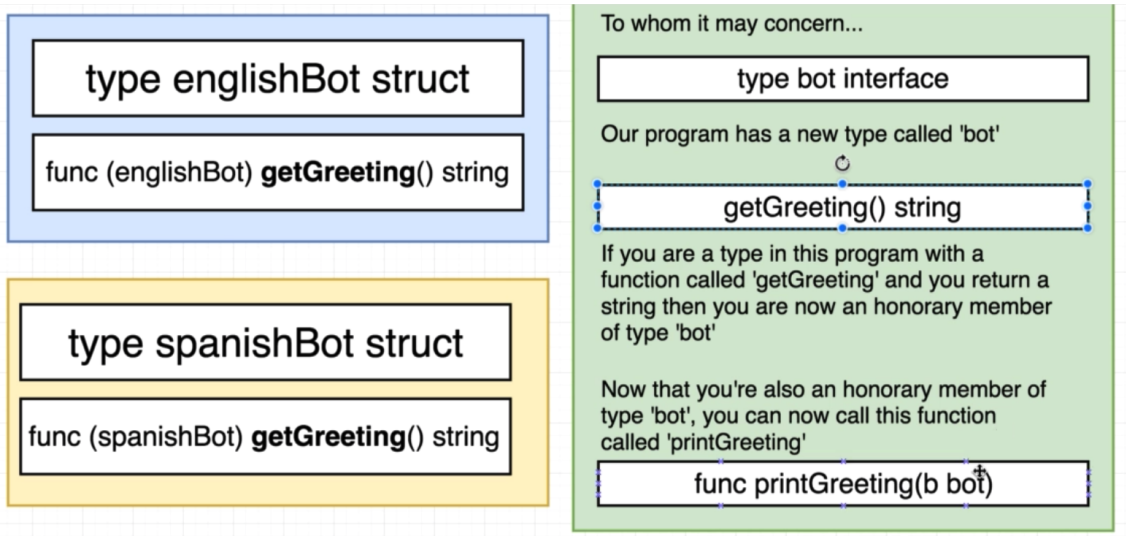
We know that: Every value has a type and Every function has to specify the type of its arguments.

So does that mean... Every function we ever write has to be rewritten to accommodate different types even if the logic in it is identical.

So one problem is that if we want to have the same function but we need to send a different type for each time, then we will need to define a new function with the same logic for each input type

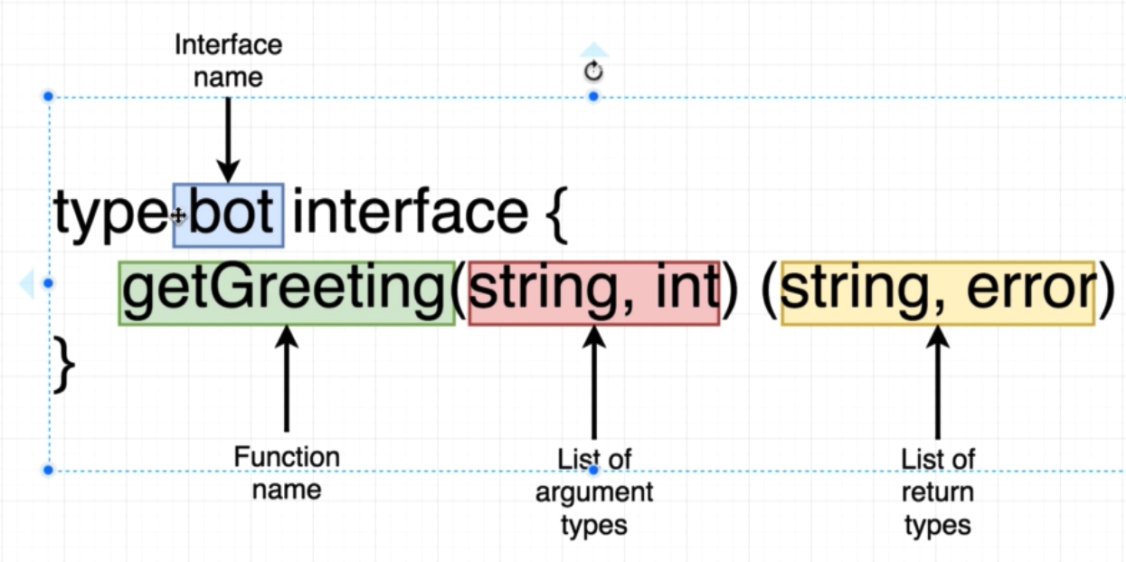


To solve this issue, we will define the function once, create an interface and pass the interface to it instead of passing the other types to it.




```
package main

import "fmt"

type bot interface { // This defines a new type from interface
  getGreeting() string // This means any type with a function getGreeting and return a string then you are now also of type b
}

type englishBot struct{}
type spanishBot struct{}

func main() {
  eb := englishBot{}
  sb := spanishBot{}

  printGreeting(eb)
  printGreeting(sb)

}

func printGreeting(b bot) {
  fmt.Println(b.getGreeting())
}

func (englishBot) getGreeting() string {
  // VERY custom logic for generating english greeting
  return "Hi There!"
}

func (spanishBot) getGreeting() string {
  // VERY custom logic for generating spanish greeting
  return "Hola!"
}
```

Defining interface should contain also the return and the arguments for the method inside it even if they are multiple returns or arguments.



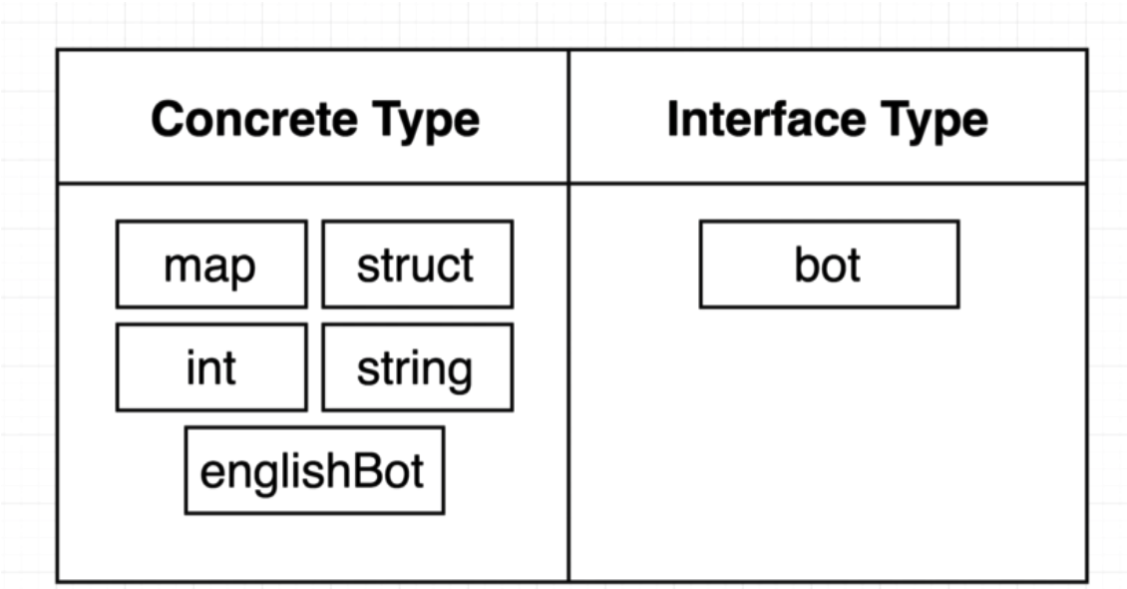
You can also define multiple methods inside in the interface:

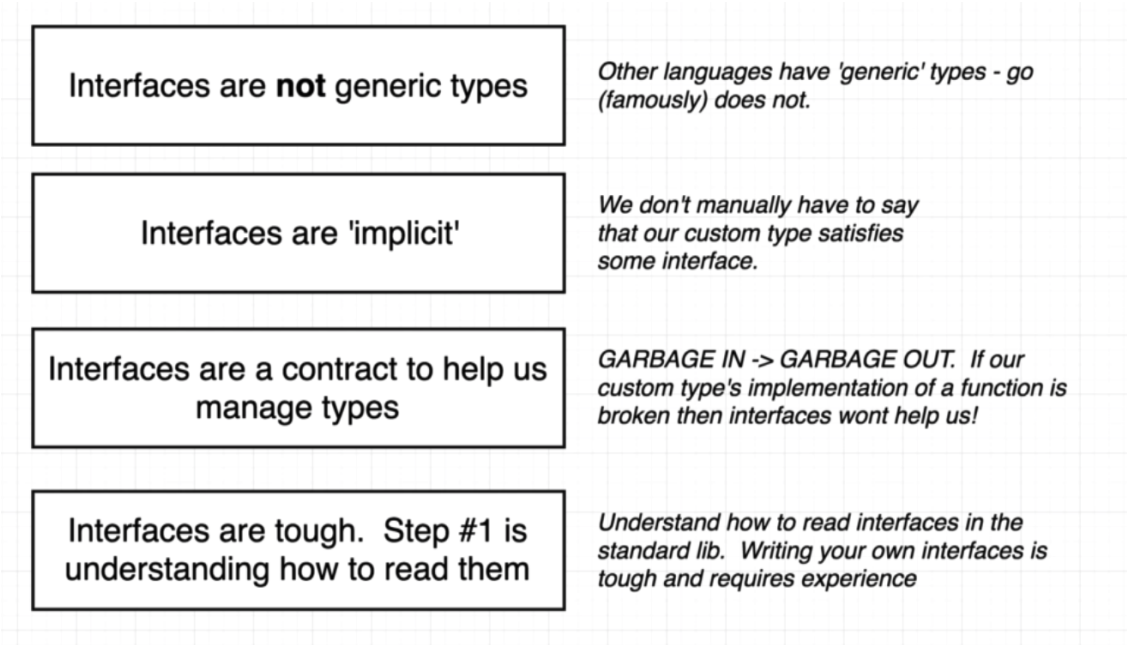
```
type bot interface {
    getGreeting(string, int) (string, error)
    getBotVersion() float64
    respondToUser(user) string
}
```

There are 2 types in Go:

Concrete type: is something that we can actually kind of create a value out of it directly and then access it, change it or edit it. (englishbot is an example of the new type)

Interface type(ex: bot): Cannot create a value directly with value type bot because it is out of type interface.

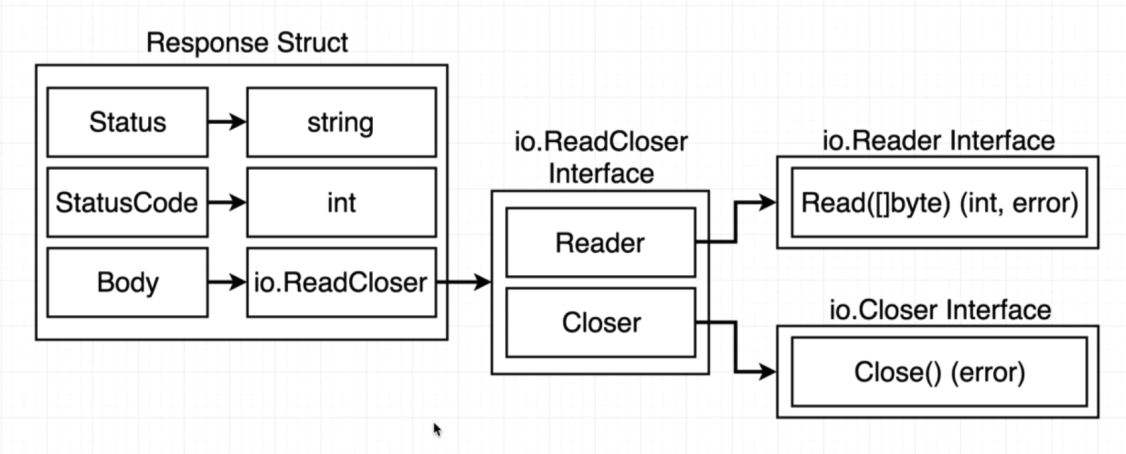




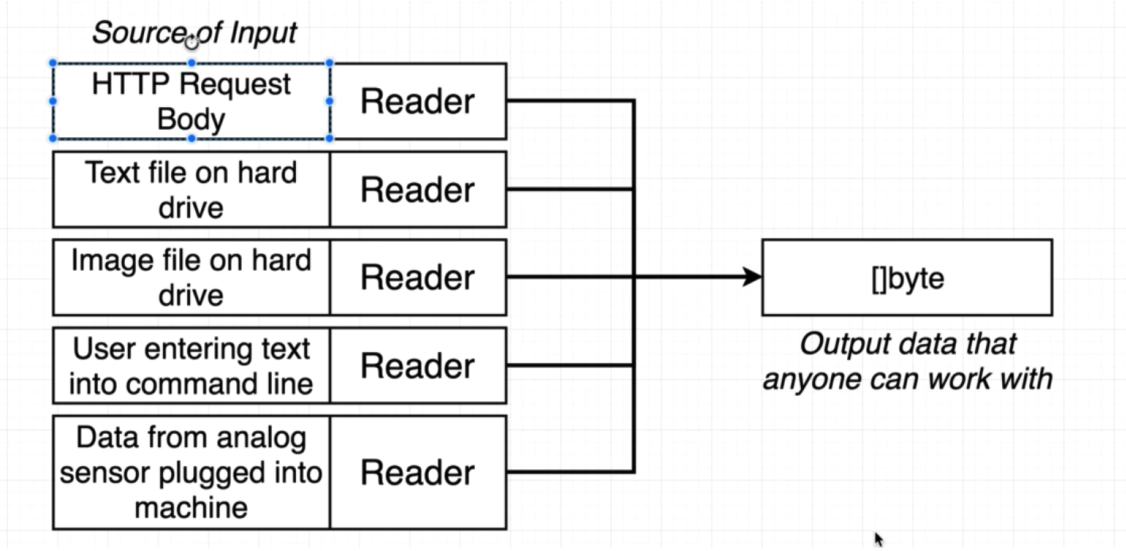
Interfaces are implicit: We just define the interface and the different functions that it expects to see then go essentially took it from there. So we don't write manual code to link between ex: (bot (interface) and english bot)

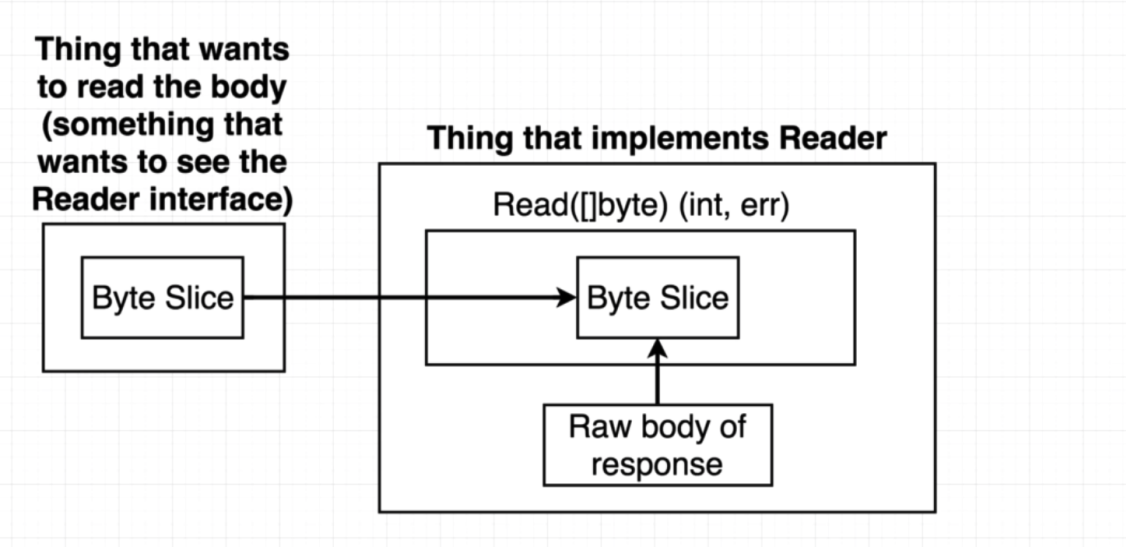
The HTTP package:

The HTTP net package returns response, and this diagram shows how the response is structured, Body is an element inside struct its type is `io.ReaderCloser` interface, and it includes 2 other interfaces `Reader` and `Closer`. So If you want to satisfy the requirements of the `ReadCloser` interface, then you have to satisfy the requirements of `Reader` and `Closer` interfaces.

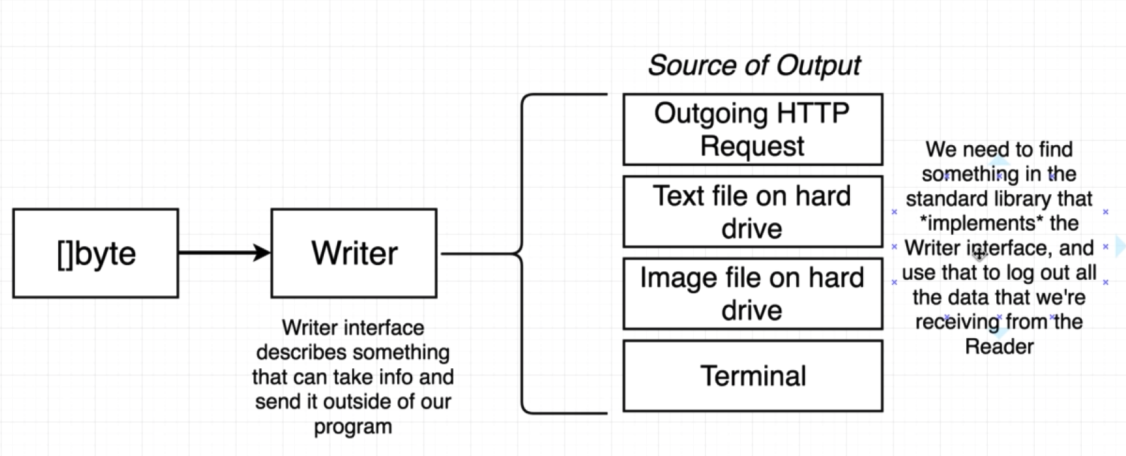


So Reader outputs the data in byte slice that anyone can read it.



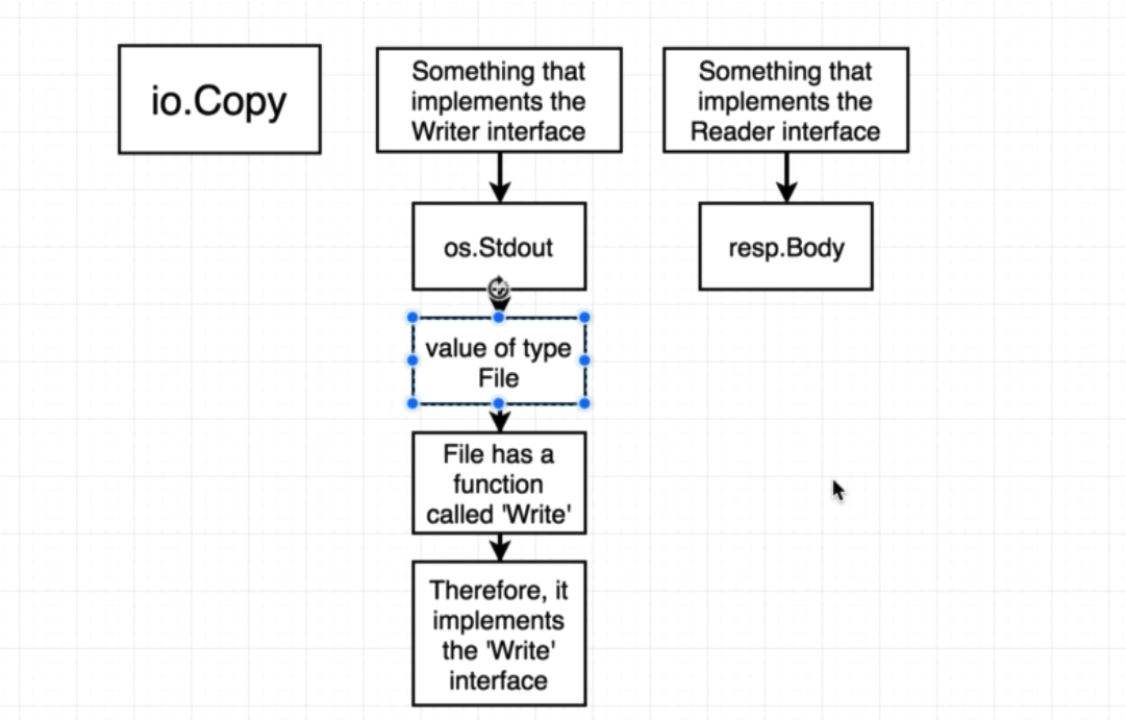


The writer interface does the opposite thins, It receives byte slice and send the output to an output channel like: Terminal, Text file, ...etc.



And the function copy expects inputs from writer (something that is taking data and sending it outside of the application) and reader interfaces. So the Copy function takes the data from outside the application and write it or copy it all out to outside channel.

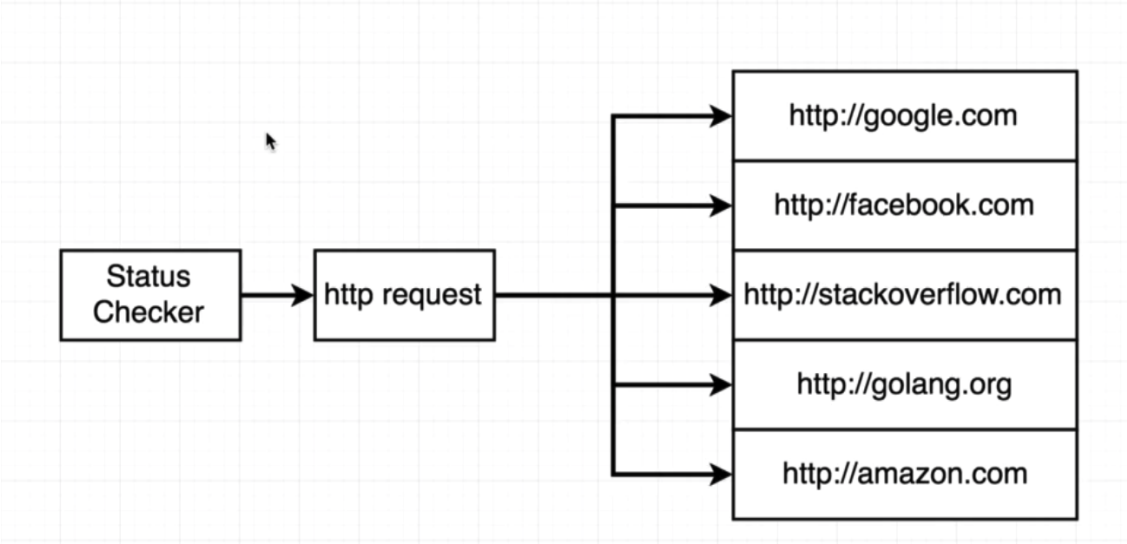
```
func Copy(dst Writer, src Reader) (written int64, err error)
```



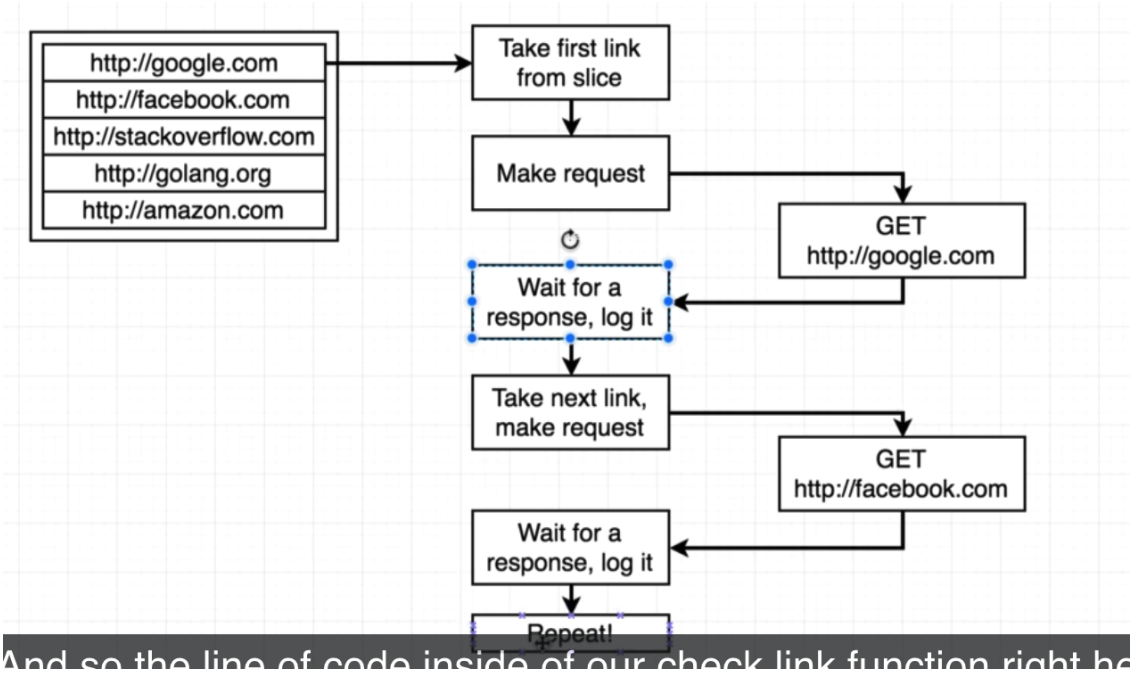
Channels and Go routines:

channels and go routines are both structures inside of go that are used for handling concurrent programming.

We will build a program called status checker, this program will take list of of very common websites and makes HTTP requests to each of these sites. And so

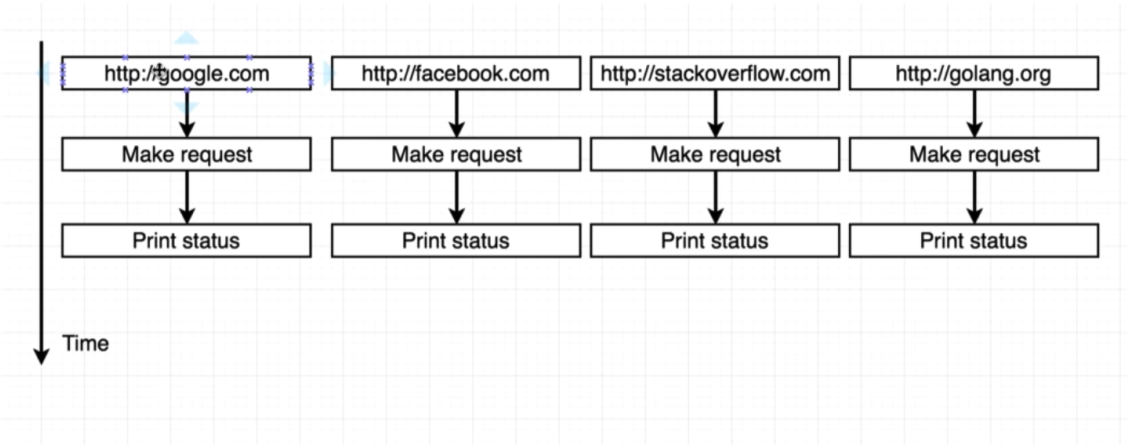


This is how the program usually will run:

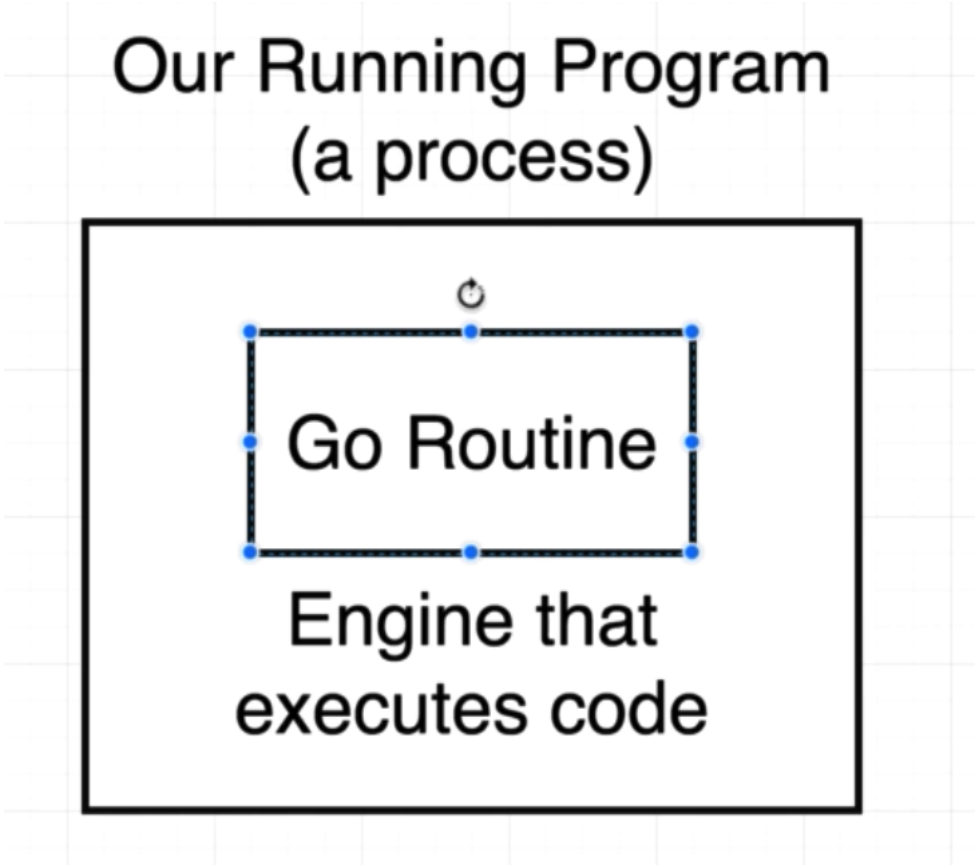


And so the line of code inside of our check link function right he

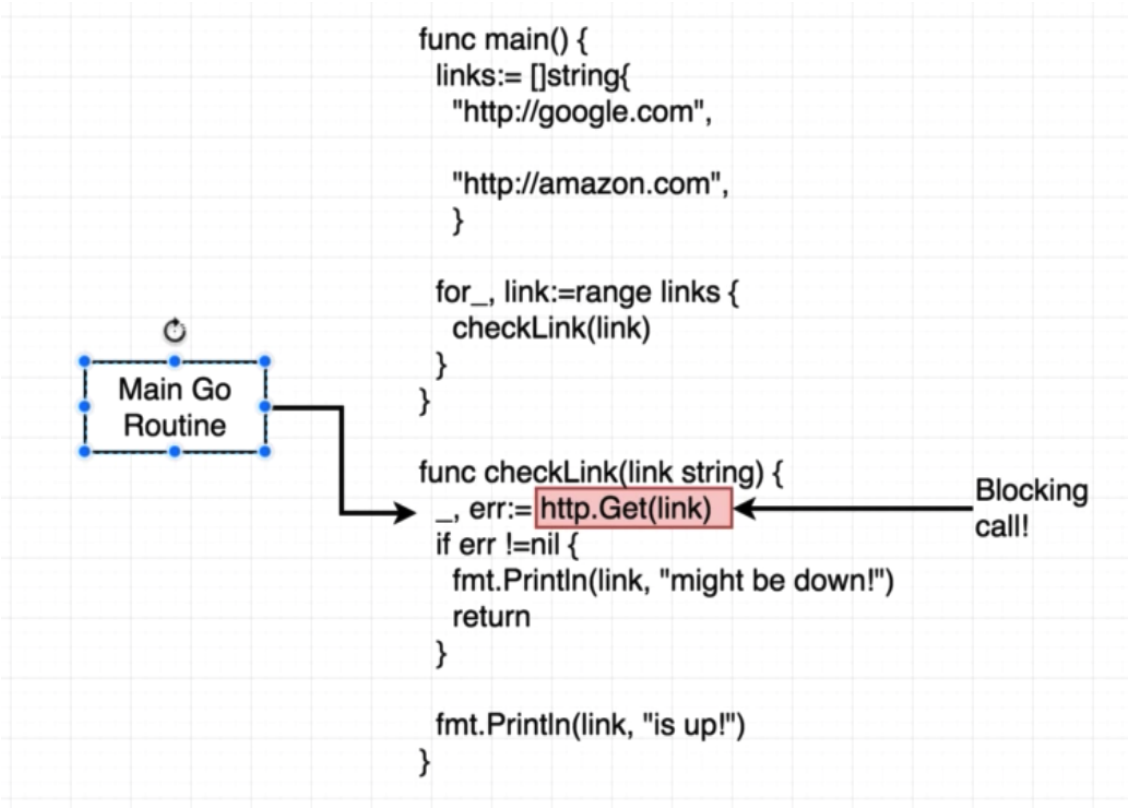
instead it's better to make the requests in parallel in the same time instead of waiting for each one of them to run.



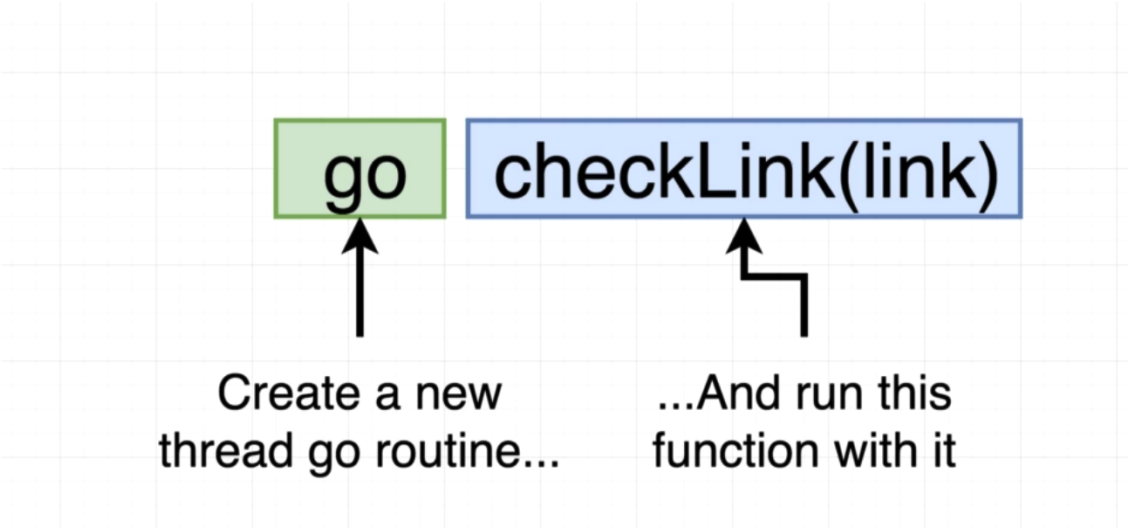
Whenever we launch a go program, like when we can pilot and execute it, we automatically create one go routine. Go routine is something that exists inside of our running program or process. This go routine takes every line of code inside of our program (the compiled file of the code) and executes them one by one.



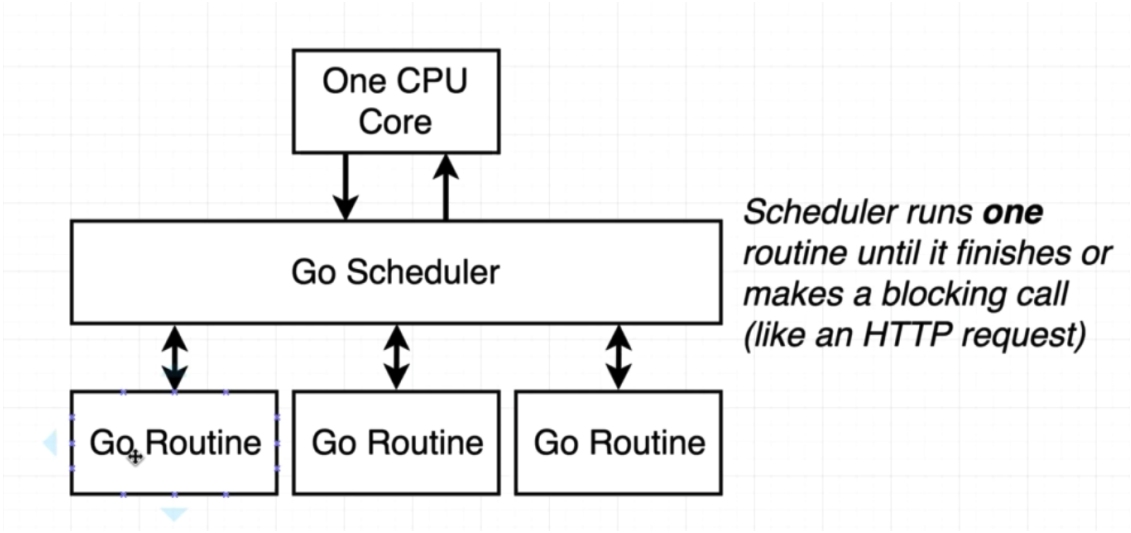
So, when executing our code, We refer to `http.Get` as a blocking call because the code inside of here takes some amount of time to execute while `http.Get` is being executed. The main go routine can do nothing else. So it's essentially frozen on this line of code right here and it has no ability to continue on or do anything else inside our program.



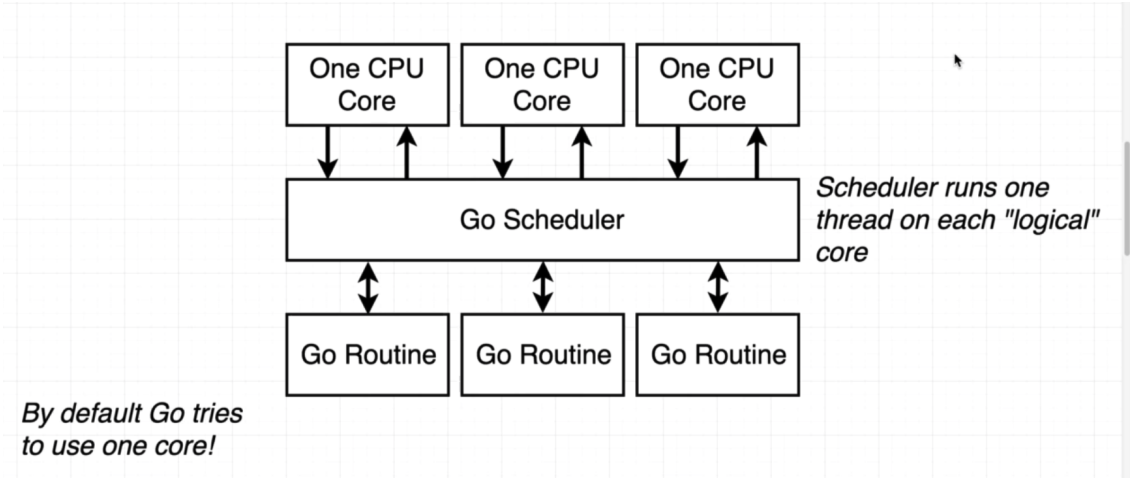
So we will solve this by introducing new routines. We will add `go` before the `checkLink` function, this means to run the function inside a new go routine.



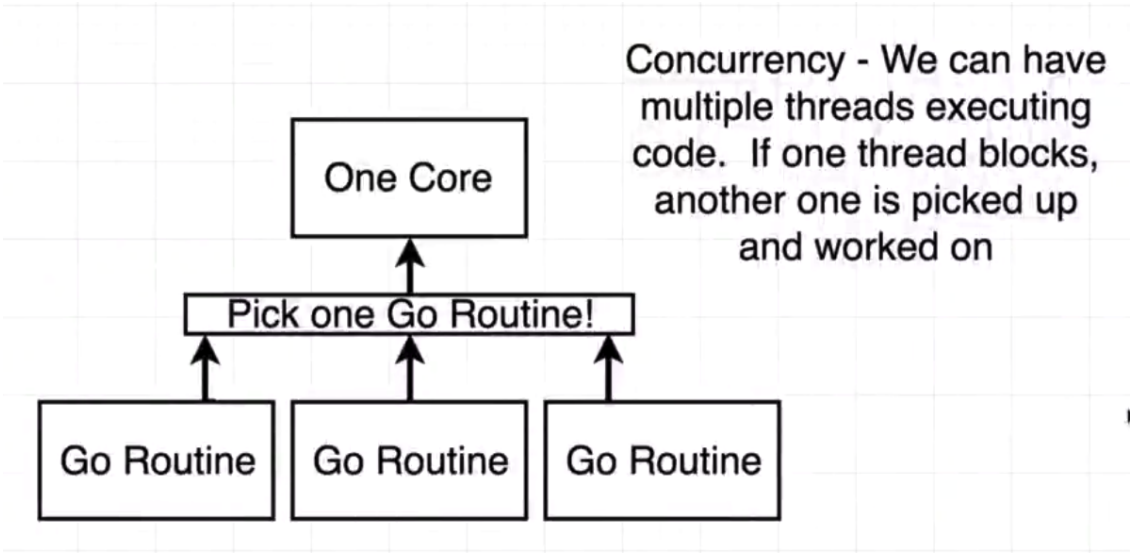
How go routines work? There is something called Go Scheduler, It works one one CPU Core on local machine. Even though we are launching multiple go routines, only one is being executed or running at any given time. So the purpose of this scheduler is to monitor the code that is running inside of each of these go routines. As soon as the scheduler detects that one routine has finished running all of the code inside it. So essentially all the code inside of a given function or when the scheduler detects that a function has made a blocking call like HTTP request we are making then it say OK, then scheduler pause this Go Routine and will start executing another go routine. So essentially, even though we are spawning multiple Go Routines, they are not actually being execute truly at the same time, whenever we have one CPU. The situation will be different when we have multiple CPU's in our local machine.
So, by default Go tries to use one core!



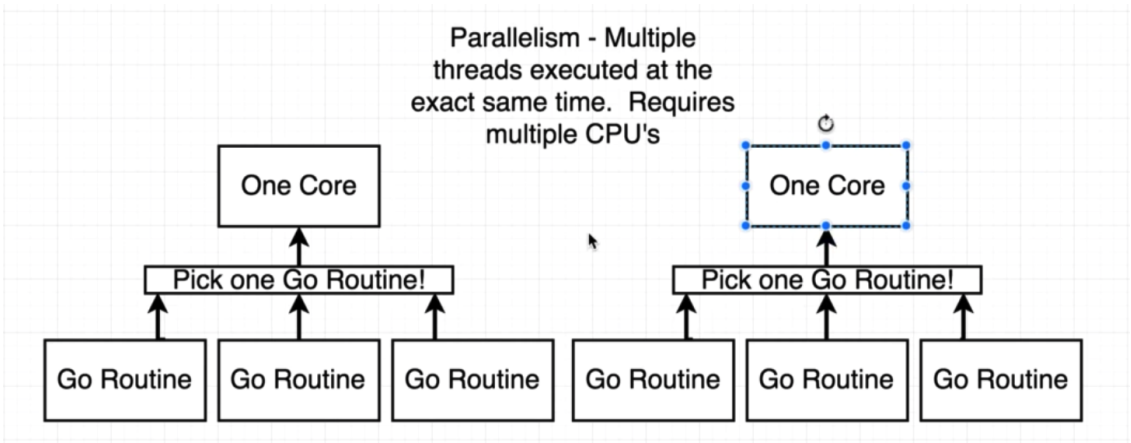
This will happen when have multiple CPU's:



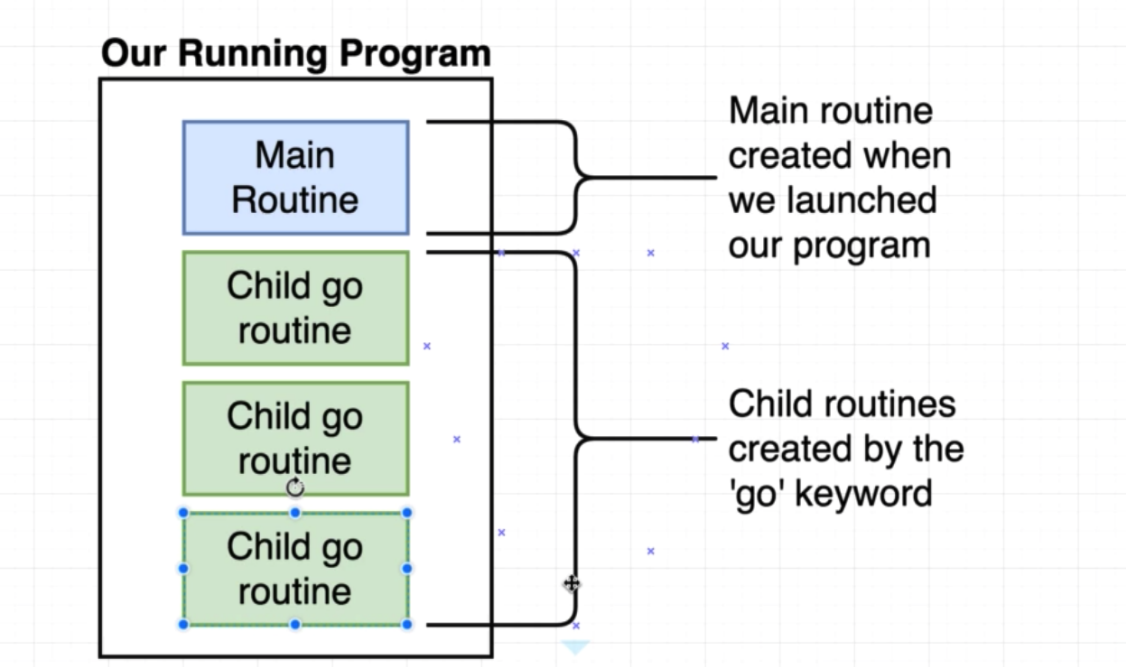
Concurrency: Program is running concurrently means that the program has the ability to load up multiple Go routines at a time. All these Go Routines might still only be running on one single core. So when we say something is concurrent, we are simply saying that our program has the ability to run different things kind of at the same time, but not really at the same time because when we have one core, we only picking one go routine. So all we're saying with concurrency is that we can kind of schedule work to be done throughout each other.



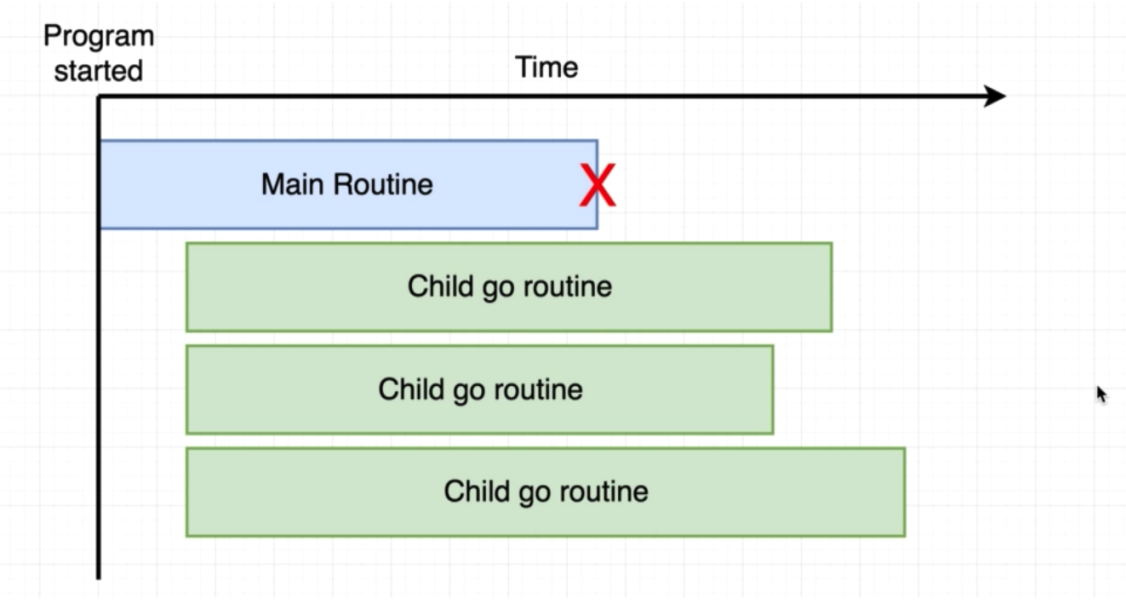
Parallelism: We only get parallelism once we start to include multiple physical CPU cores on our machine, we can literally say that we can do multiple things at the exact same time



When we first start up our program, we get this single main go routine. And then anytime we use that go keyword, we are creating child routines.



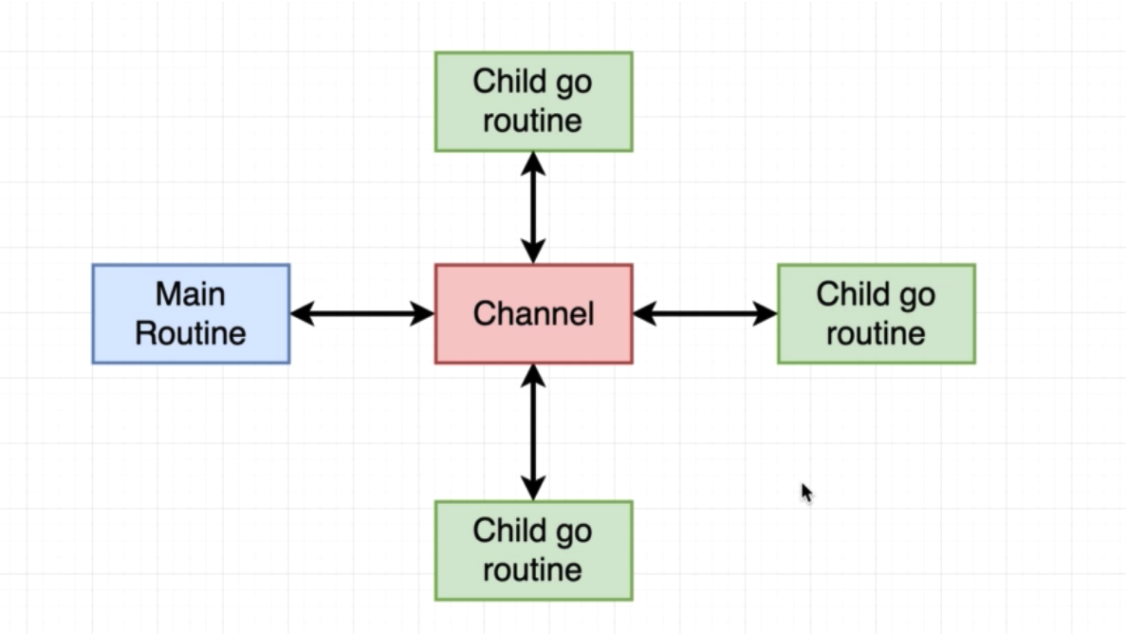
The main routine is the singe routine inside of our program that controls when our program exists or quits. So when we first start up our program, we get this main routine created by default. So the main routine starts running at some point in time as we start enter the for loop. Once main routine finishes iterating over the loop, it exits and it doesn't care if the children routines still running or not.So to handle this, we will need to use something called channels.



Channels:

Channels are used to communicate in between different running Go routines. We are going to use a channel to make sure that the main routine is aware of when each of these child go routines have completed their code.
So essentially, we're going to create one channel and that channel is going to communicate between all of these different go routines. The channels are the only way that we have to communicate between go routines. Channel are of specific types so when we define a channel of string, we can't pass integer to it.

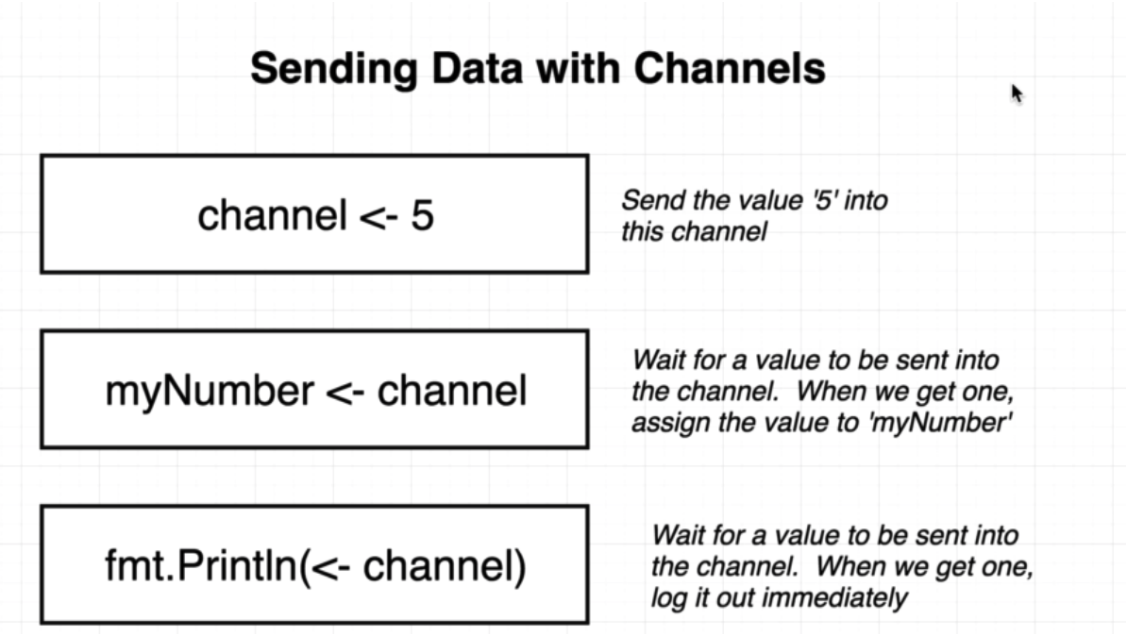
We can think of channel like this shape:



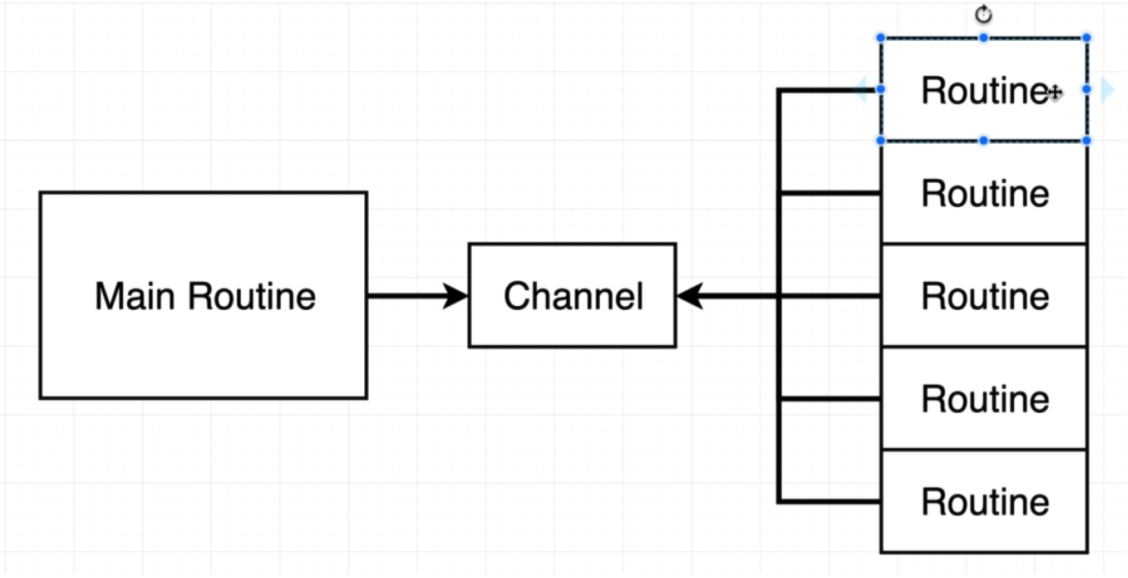
Defining channel:

```
c := make(chan string) // This is how we create new channel. make is a built-in function that will create a value out of the g

  for _, link := range links {
    go checkLink(link, c) // We pass the channel to the Go Routine definitions that will use it
  }
```

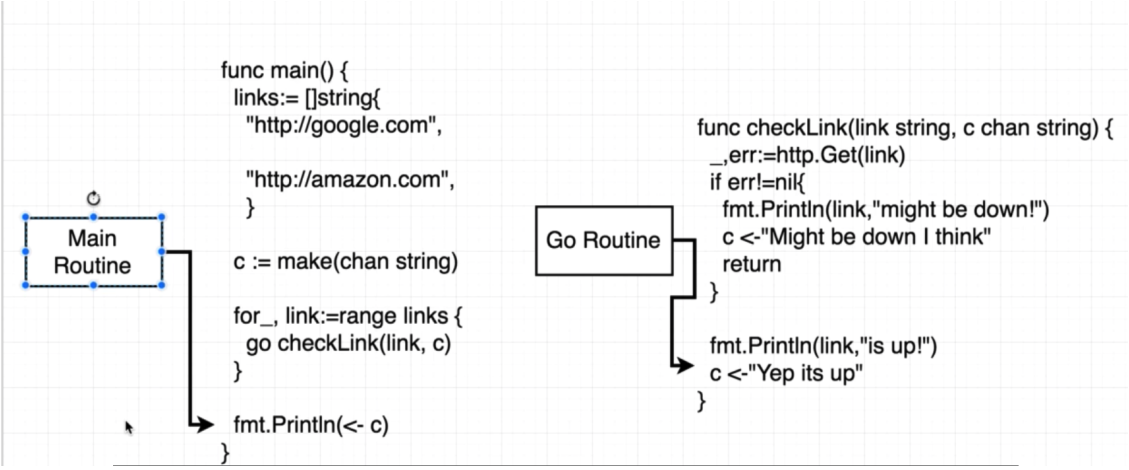
Sending Data with Channels:

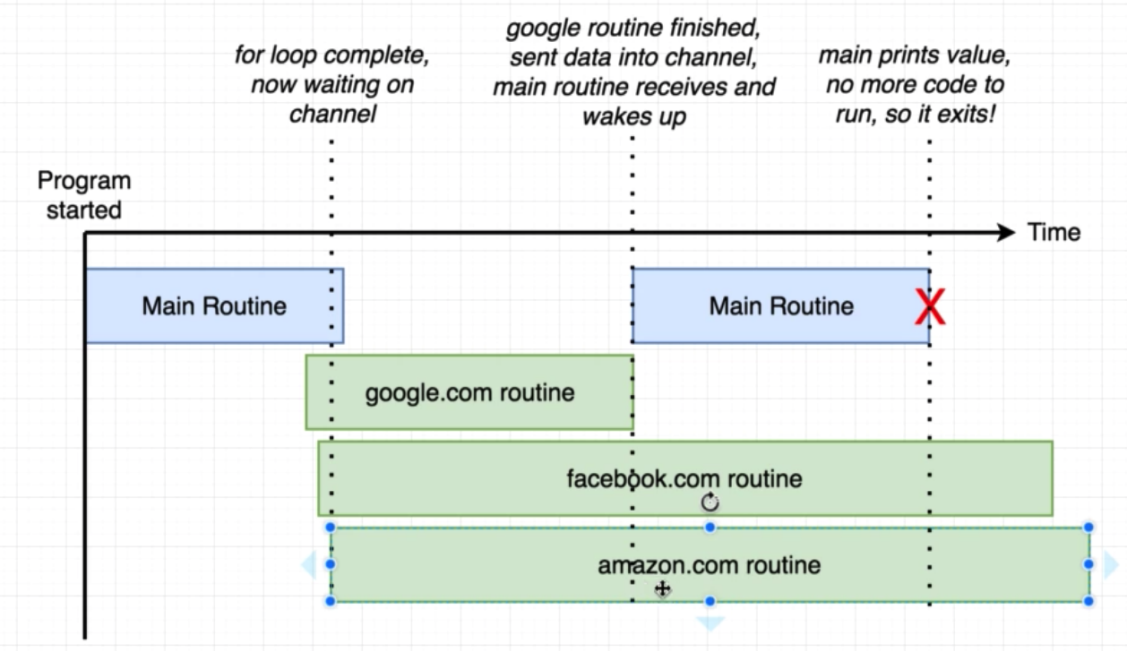


We might want to send data from main routine to Go Routine or the reverse:

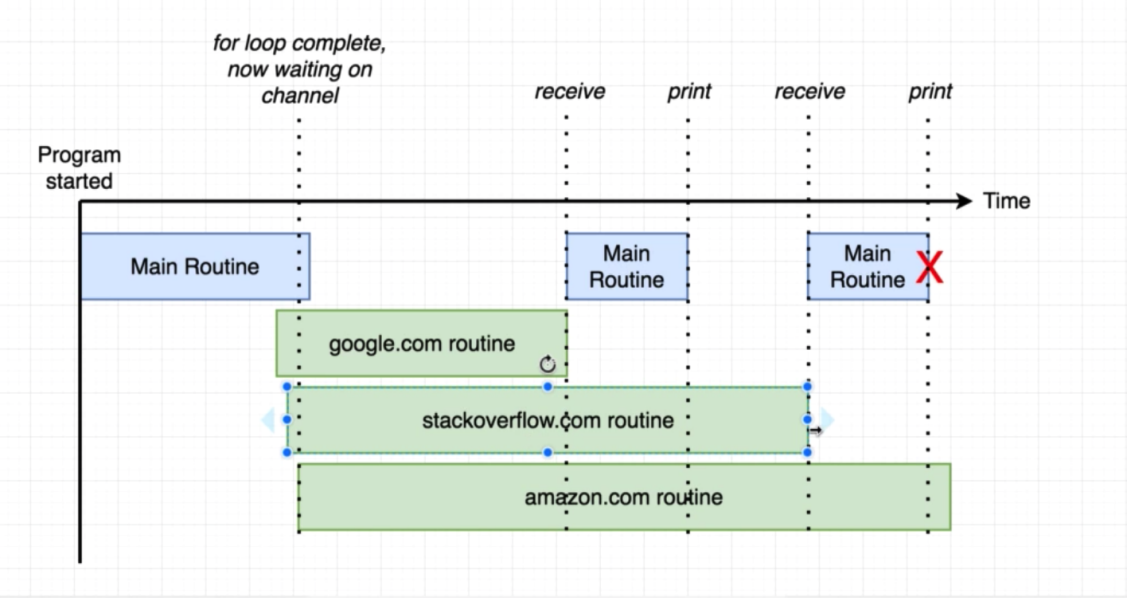


Receiving message from the channel is a blocking function, we have to wait for message to come through before the run time or before this routine is going to continue on passes this line:



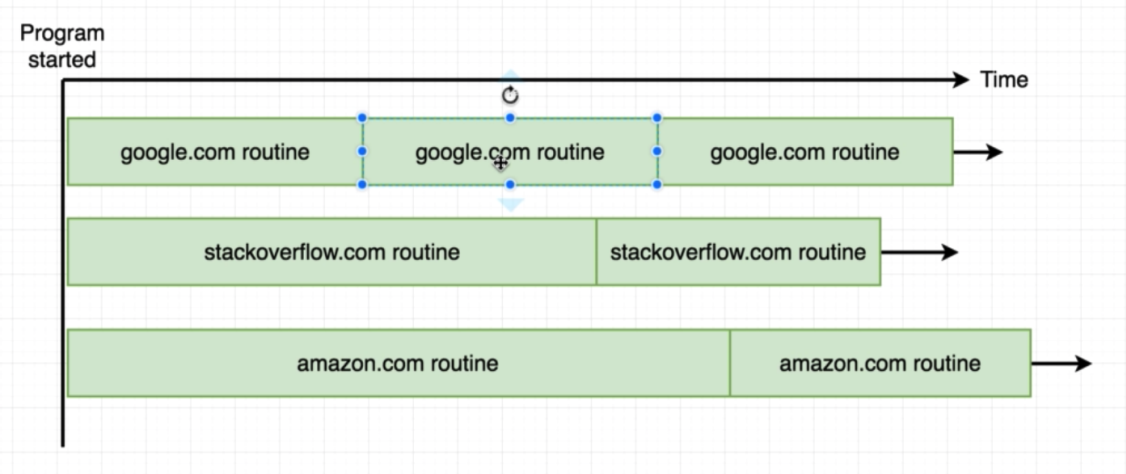


Fix for the previous is we will add print for each routine:



Repeating Go routines:

If we want to make the routine running repeatedly, for example: whenever the routine for google finishes up will make another request.



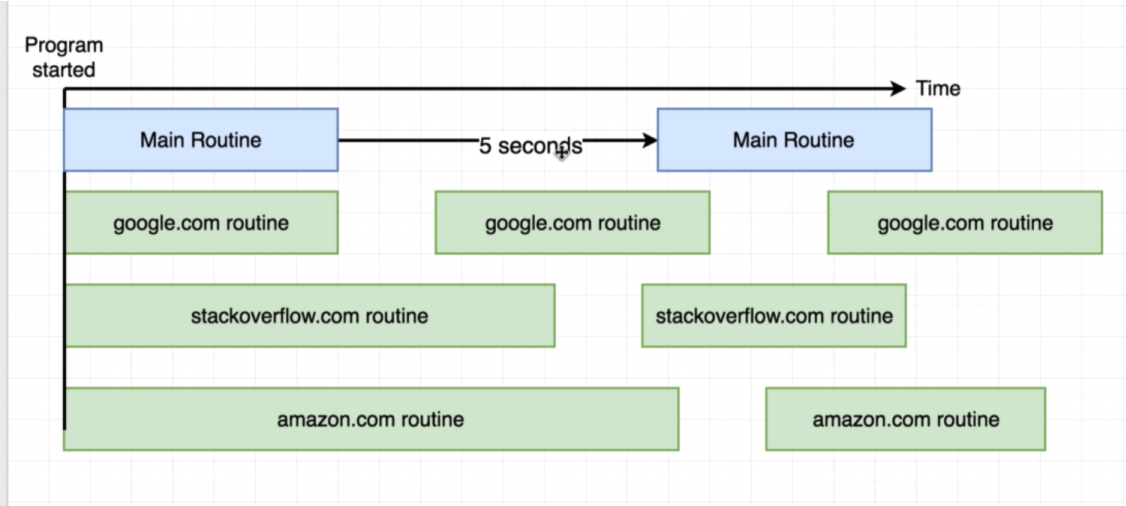
So instead of pushing a string to a channel we will push the link to the channel, and whenever we receive the link, we will kick-off another Go routine.

Sleeping a Routine:

For the repeated Routine, we need to add pause before repeating the Go routine.

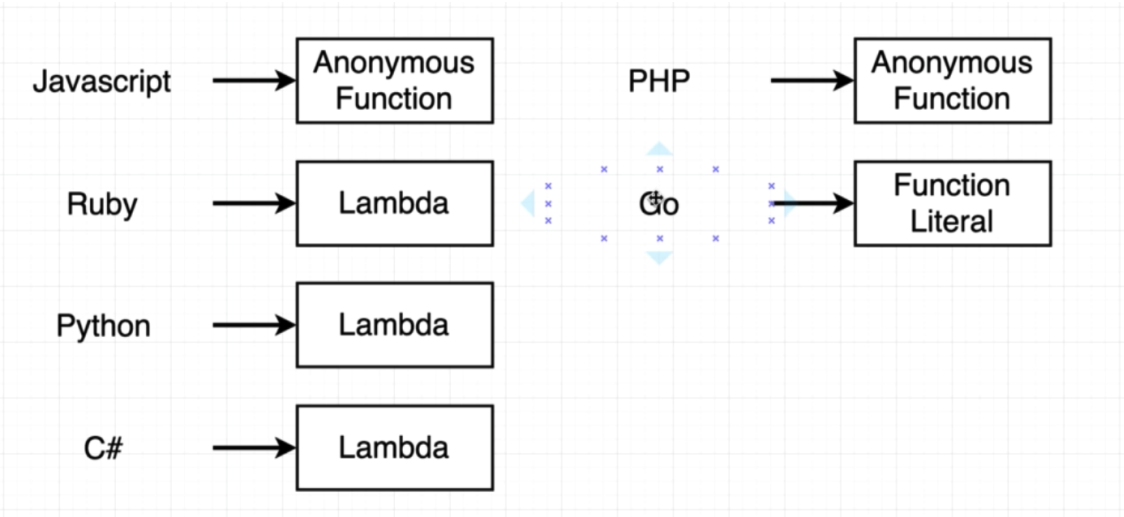
```
timer.Sleep(5 * timer.Second)
```

Adding this sleep in the main routine, that is a blocking call. and it means that while the main routine is paused, it cannot receive any other messages through the channel. Those messages will not be lost. They just kind of like lined up or queued up. But this is that we are saying that the most we can only ever execute one new Go routine once every 5 second because when we put this time that sleep statement inside of main routine, we're saying Go routine for the first request just finished up right here and we will pause the main routine. and we don't wanna add it to the `checkLink` because this function supposed to get the link immediately. Like the below diagram:



Literal Function:

So we will use function literal, which is an unnamed function that we use to wrap some little chunk of code o we can execute it at some point in the future.




```
// This is how to define literal function
go func(link string) {
      time.Sleep(5 * time.Second)
      checkLink(link, c)
   }(l) // these are the paranthese that actually execute literal function
```