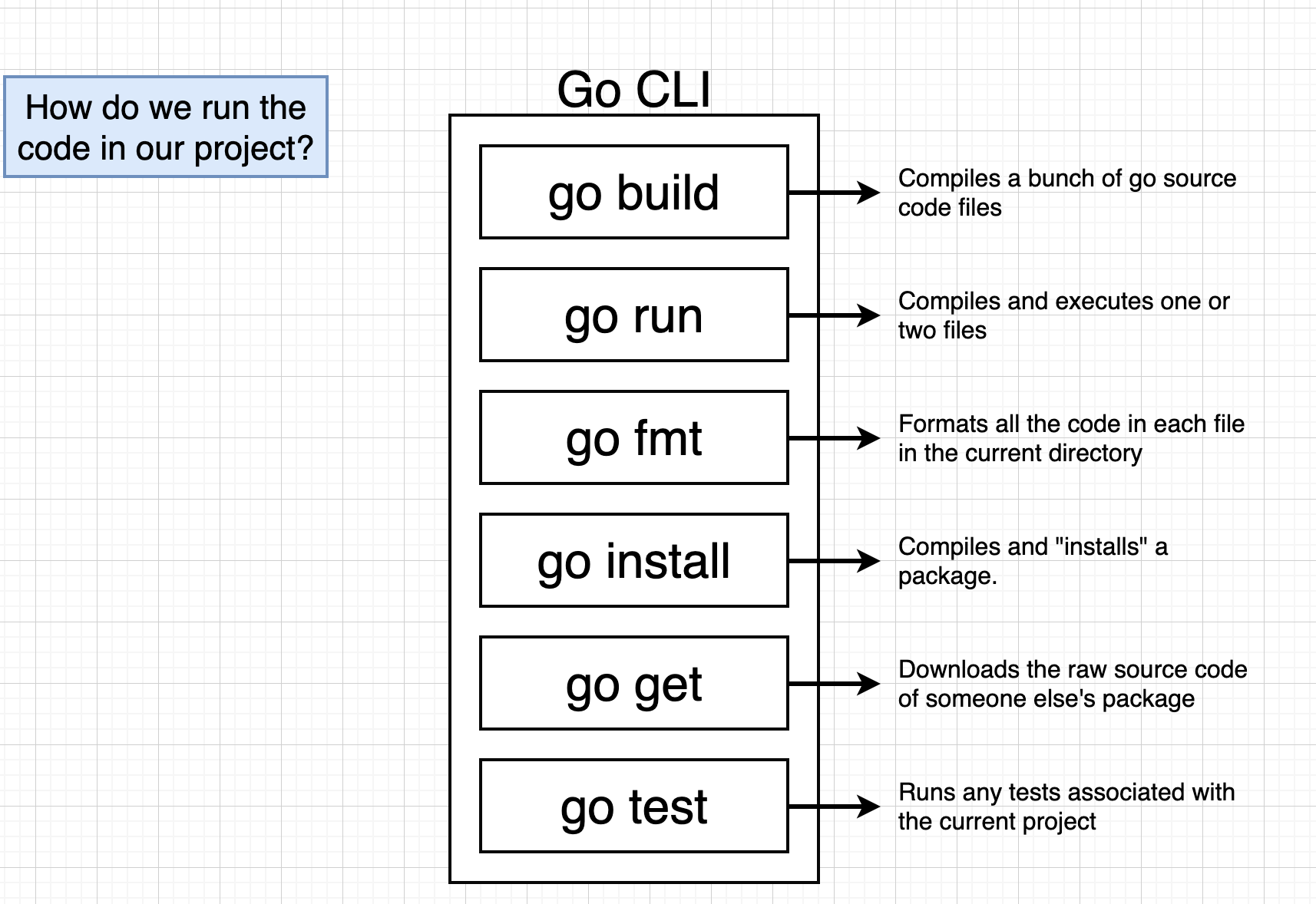
**GO Commands**



* Go run is used to compile and immediately execute the program.
* Go build only compiles the program and doesn’t execute it.
* Go install and go get are used to handle dependencies in our project (use code written by someone else to get access to it on my projects).

**Packages**

Each file must declare the package name to which it belongs at the top (package name and directory name need not be the same).

Types of packages:

1) Executable: main (package name) is used to make an executable package. Any package other than main won’t spit out a runnable file when go build command is run on it. It must contain a function named main inside it as well.

2) Dependency: code used as helpers.

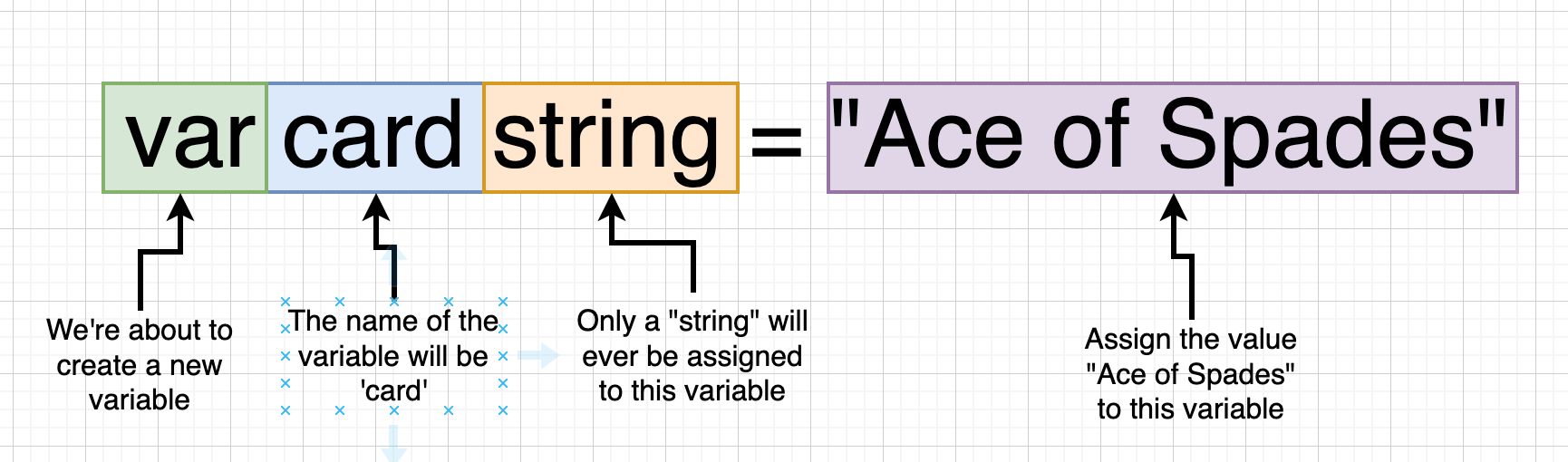
**Import Statements**

* Allows us to gain access to code/functions from other packages.
* Import “fmt” allows us to use the code and functions inside the fmt package.
* Fmt is a library package included by the go-programming language by default.
* Used for printing stuff (like debugging).

**Package Structure**

1. Package declaration.
2. Importing other packages.
3. Declaring functions.

**Variable Declarations**



We define “string” explicitly because go is a dynamically typed language.

* Basic way of defining a variable: *var card string = “Hello”*
* Relying on the compiler to figure out the data-type: *card := “Hello”*
* We can only use *:=* when creating a variable for the first time and not after that (after that =).
* We can initialize a variable and assign a value to it at a later point of time.

*var deckSize int*

*deckSize = 52*

* We can initialize a variable outside of a function but can’t initialise it at that time.

v*ar deckSize int*

*func main() {*

*deckSize = 52*

*}*

**Function**

* Basic syntax for a function returning a string:

*func newCard() string {*

*return “Hello”*

*}*

**Slices and For Loops**

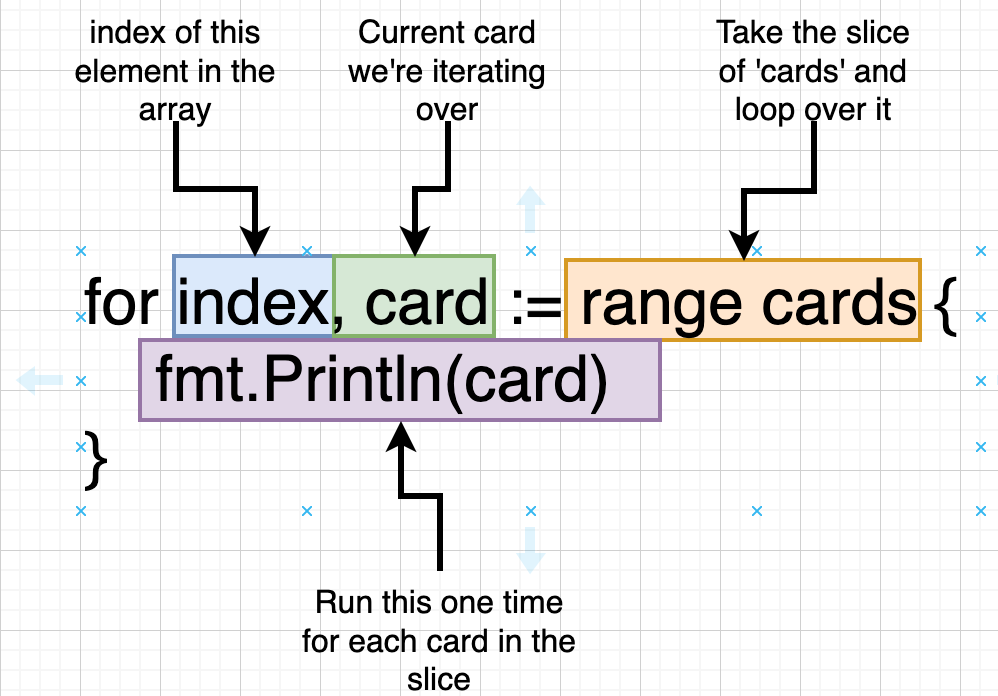
* Array is used for holding a list of records. It has a fixed length.
* Slice has more features as compared to an array. Can grow or shrink in size.
* All the contents/records must be same inside both of them.
* Basic syntax for a slice:

*cards := []string{“Ace of Spades”, “Six of Heart”}*

* *append* method takes in the existing slice as the argument, new element to be added. It doesn’t modify the original slice but returns a new slice to the variable declared on its left.

*cards = append(cards,”Six of Spades”)*

* Iterating over a slice and printing all the contents inside of it.



*NOTE: Why are we using* ***:=*** *everytime we loop over the slice (considering we only used it when initializing the variable for the first time)? This is because with each iteration we are throwing away the previous index and card declared and re-delcaring the variables.*

* Slices are 0 based indexed and we can access the element as *slicename[index]*.
* *slicename[startIndexIncluding:uptoNotIncluding]* not specifying the start index treats it as 0 and not specifying the end index treats it as end of the list.

**OO Approach vs GO Approach**

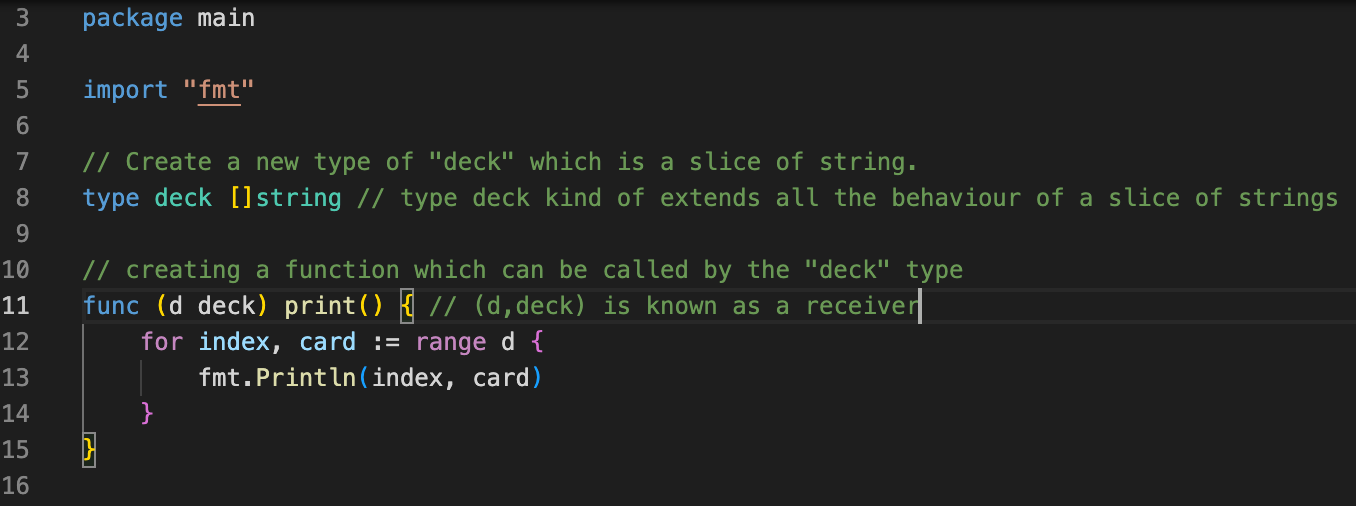
* Go is not an object oriented programming language and hence, there is no idea of classes in it.

New special type called a “deck” type. It is same as a slice of string but because we are making the extra type, we can create a bunch of functions which only work with the “deck” type.

*type deck []string*

Now, cards := []string{“Ace”,”King”} can be replaced by cards := deck{“Ace”,”King”}.

* Functions

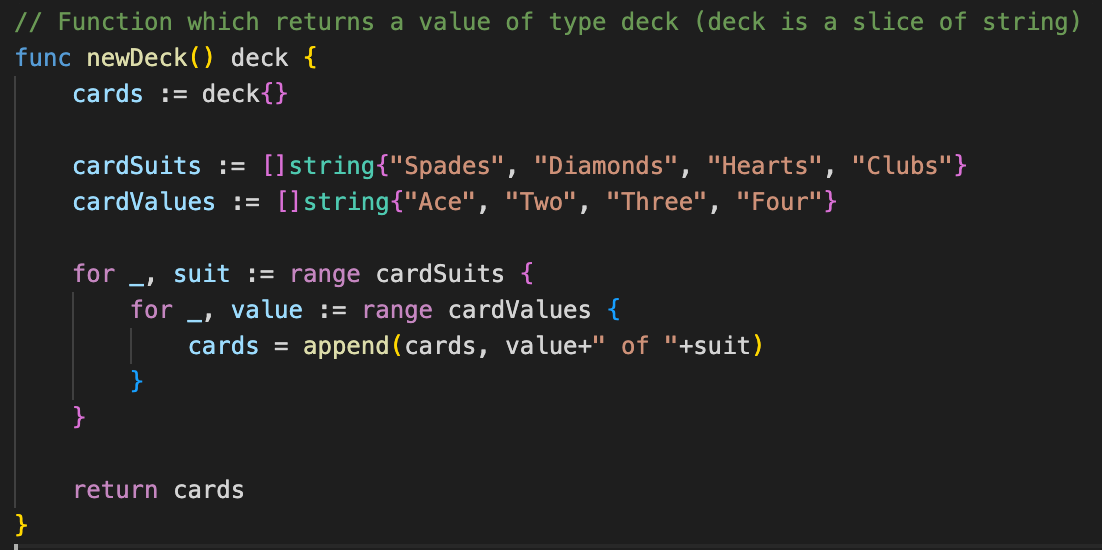


* *print()* is the name of the function.
* *(d,deck)* is the a receiver. “d” is the working variable/instance of deck we are working with.
* Any variable of type “deck” in our applications gets access to the *print* method (made possible by the receiver).
* When *cards.print()* is called, *cards* is passed as a variable to the print function (referred to as “d”).
* “d” here is the receiver and we usually name the receiver with a single/2 letters and it’s of type deck.
* Whenever a function has to return something (a value eg string) we place the annotation after the function name (eg func newDeck() deck, returns a value of type deck).
* If we don’t want to use a value then, we replace it by ‘\_’ to tell go that we won’t be using it.

for \_, card = range d {

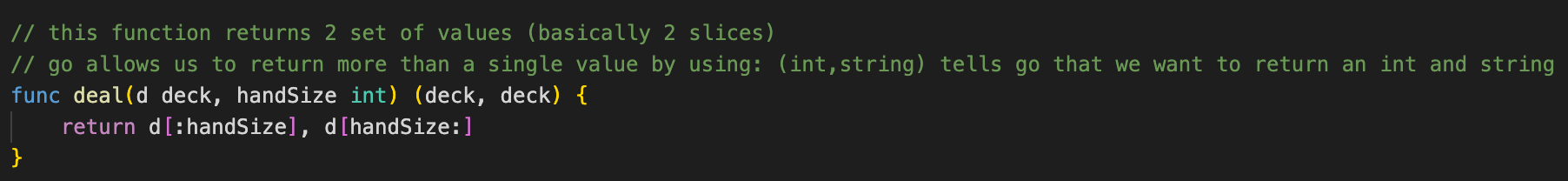
Fmt.Println(card)

}

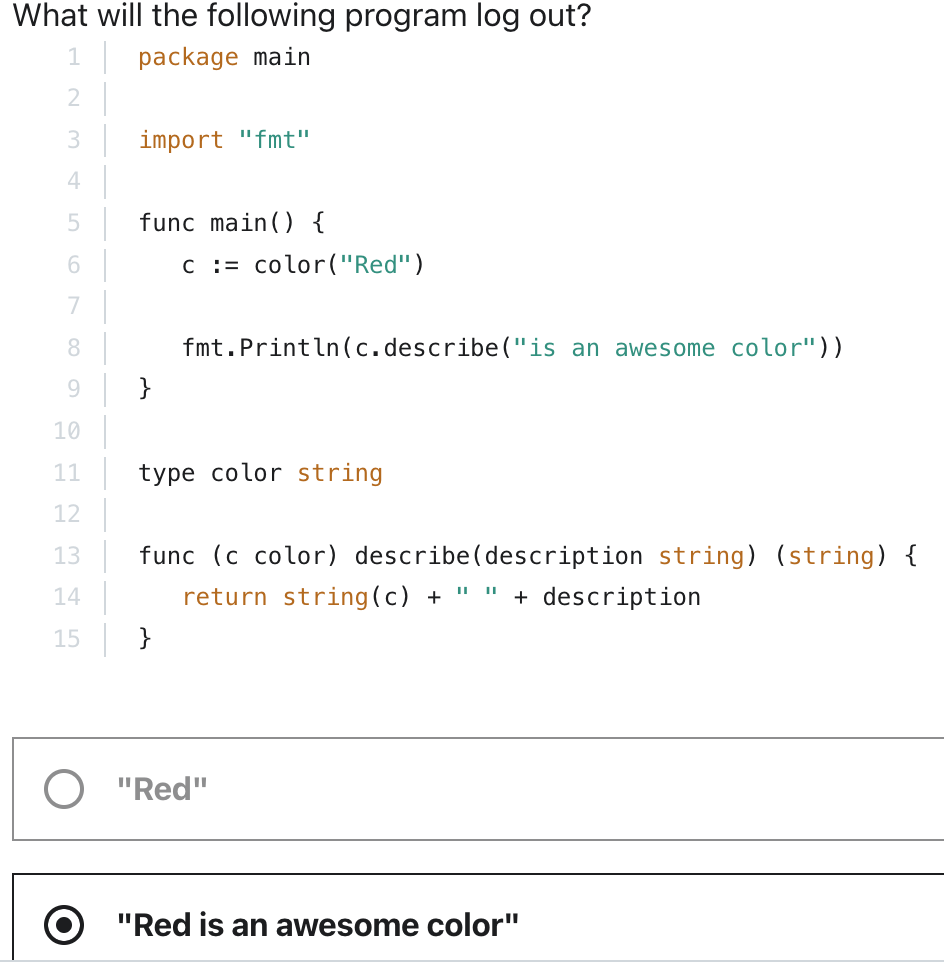


**Multiple return statements**

* + Go allows us to return more than a single value by using: *func (d deal) (int,string)* tells go that we want to return an int and string.
  + We can call a function without importing it if its present in the same package (even if its in a different go file).



\*\*\* IMPORTANT QUESTION \*\*\*

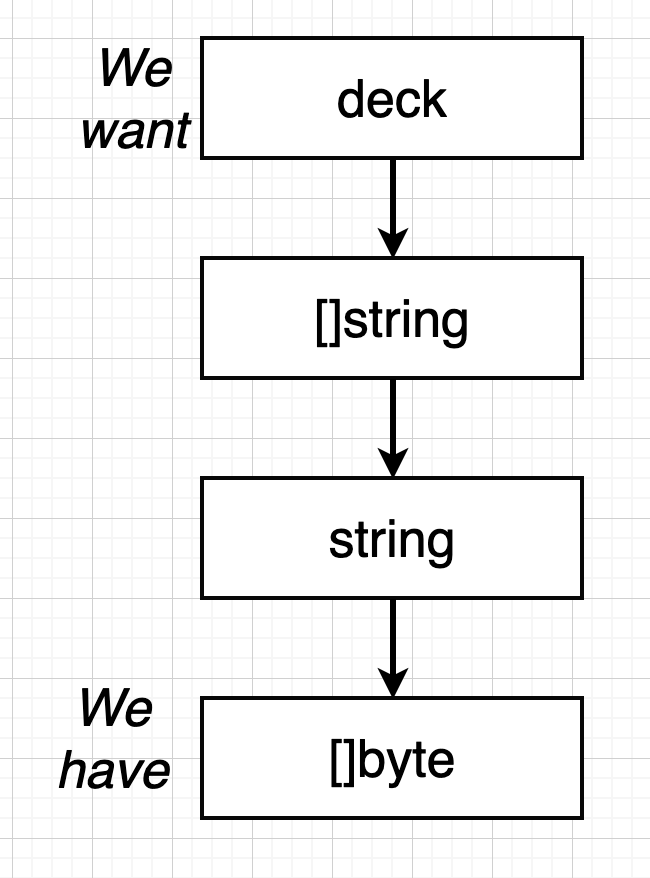


**Writing to a File (Byte Slices)**

* **func WriteFile:** func WriteFile(filename String, data []byte, perm os.FileMode) error // 3 arguments. data []byte is a byte slice. Writes data to a file with the name “fileName”. If the file does not exist, it is created with the permissions “perm” else, WriteFile truncates it before writing.
* **Byte Slice** is a string of characters. It is a slice where every element inside of it corresponds to an ASCII character. A byte slice basically represents a string (since the elements are ASCII codes corresponding to certain characters).

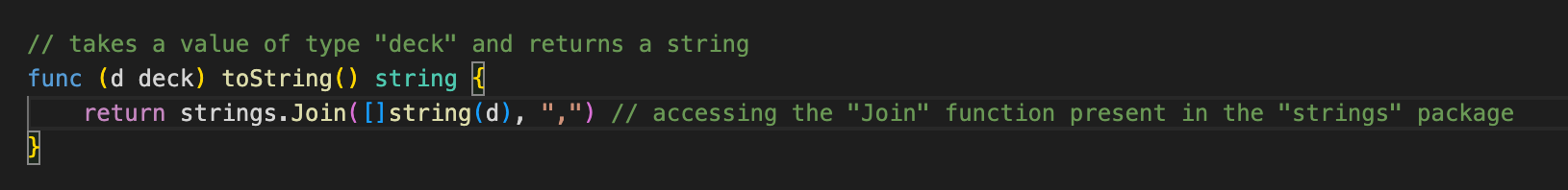
**Type Conversion**

* **TypeWewant(valueWeHave)**: *[]byte(“Hi There”)* converts the string “Hi There” to a byte slice.



We can take the slice of string and join every value inside of it with a ‘,’ using a package called “strings” and then using the function (present inside the package “strings”) “join”.

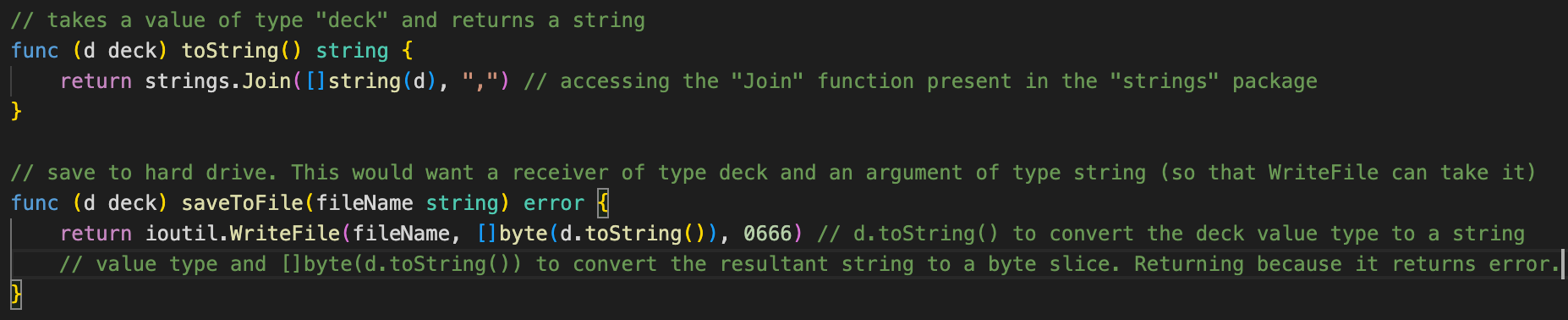
**Func Join(a []string, sep string) string** : returns a string (sep is the seperator).



*fmt.Println(cards.toString())* in main

**Saving the Data to the Hard Drive**

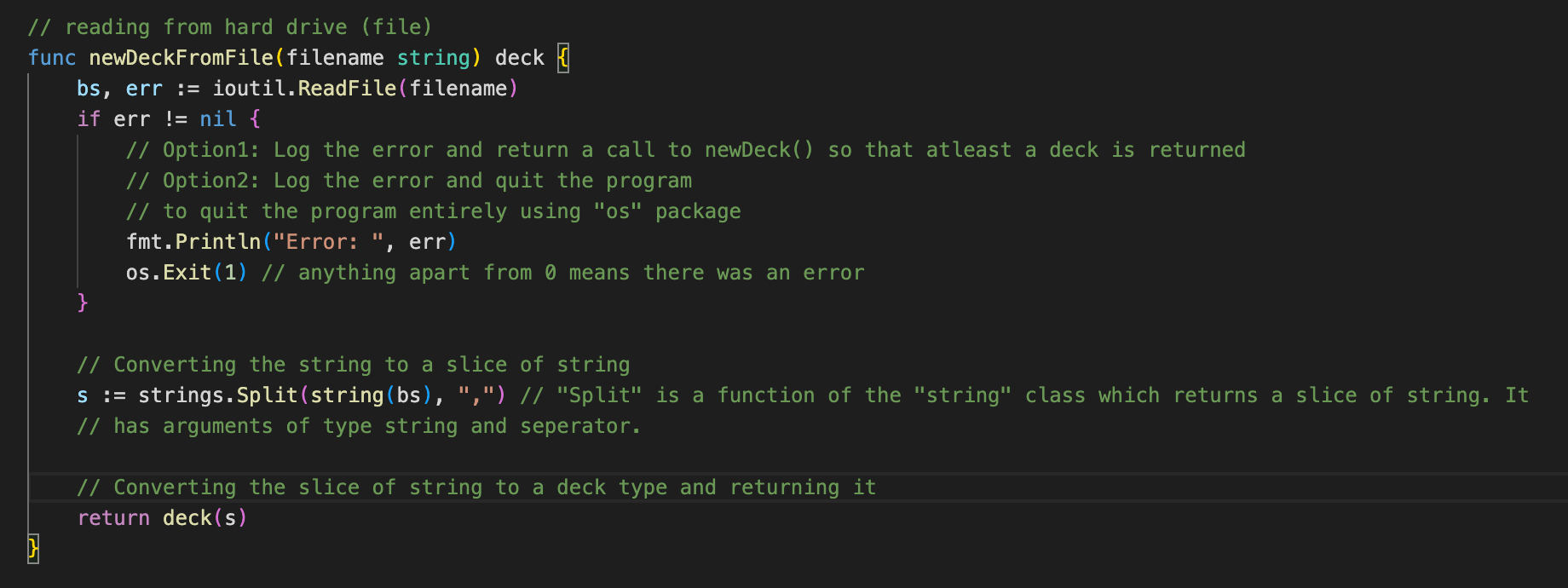
* **func WriteFile:** func WriteFile(filename String, data []byte, perm os.FileMode) error
* WriteFile function is a part of the ioutil package.



cards.saveToFile("my\_cards") // in main

**Reading from a File**

* Use the function **ReadFile(filename string) ([]byte, error)** takes in a filename and returns a byte slice (string of characters which is a list of cards seperated by comma) and an error object (value of type error).
* We have to convert the slice of byte(as stored in the file) -> string (comma seperated) -> list of string -> deck.
* **func Split(s string, sep string) []string** : takes a string and a seperator as an argument and returns a slice of string.



**In main**

cards2 := newDeckFromFile("my\_cards")

cards2.print()

**Shuffling the deck of cards**

for each index, card in cards

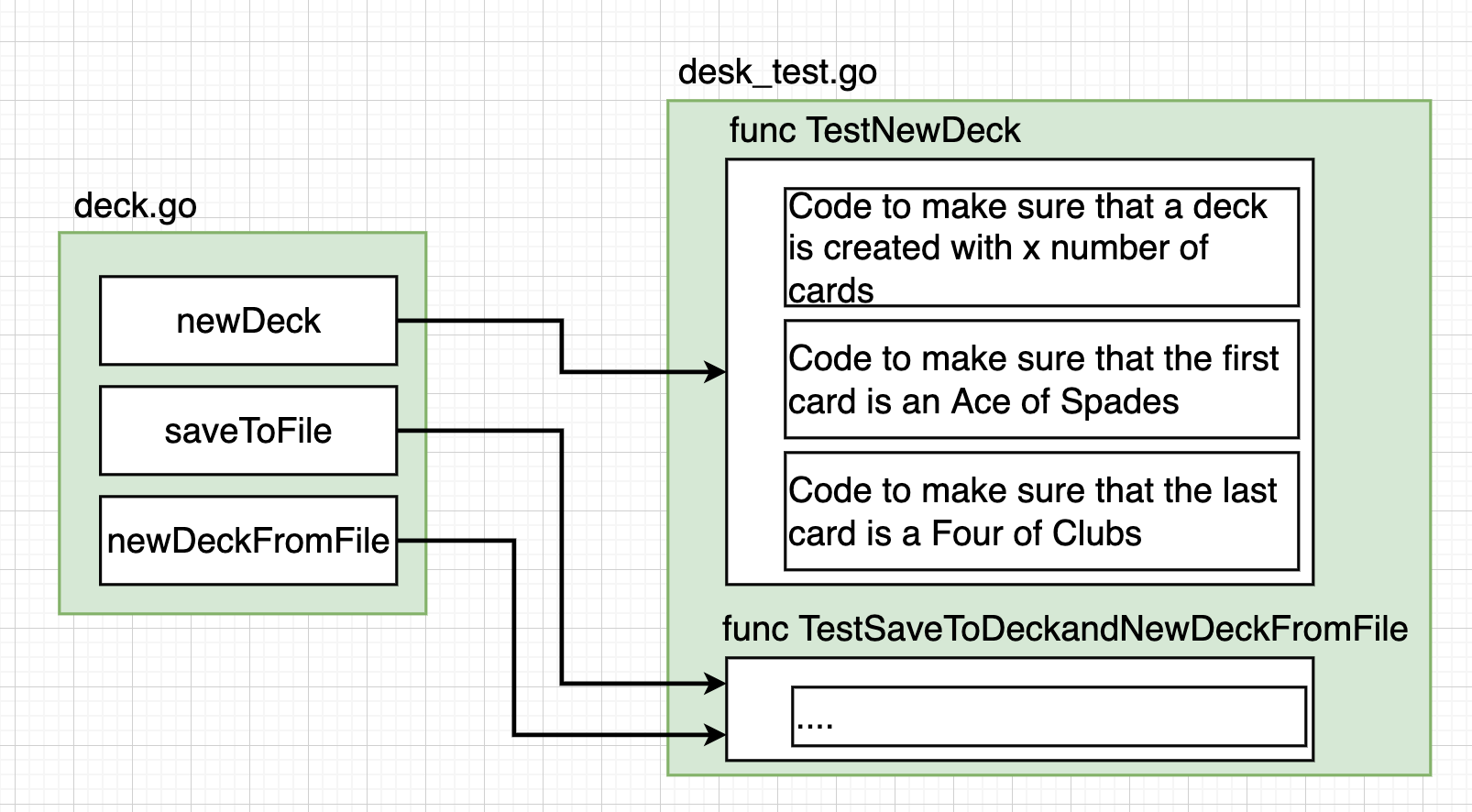
generate a random number between 0 and len(cards) - 1

swap the current card and the card at cards[randomnumber]

* **func Intn (n int) int** : generates a random number between 0 and n (not inclusive) and returns it. However, this is a psuedo-random generator which uses a particular seed value (source of randomness) which is always the same and hence we get the exactly same list of random numbers.
* Use this <https://www.udemy.com/course/go-the-complete-developers-guide/learn/lecture/7797304#search> to generate random values (our own random generator).

**Testing in GO**

* To make a test, create a new file ending with **\_test.go** and run **go test**.

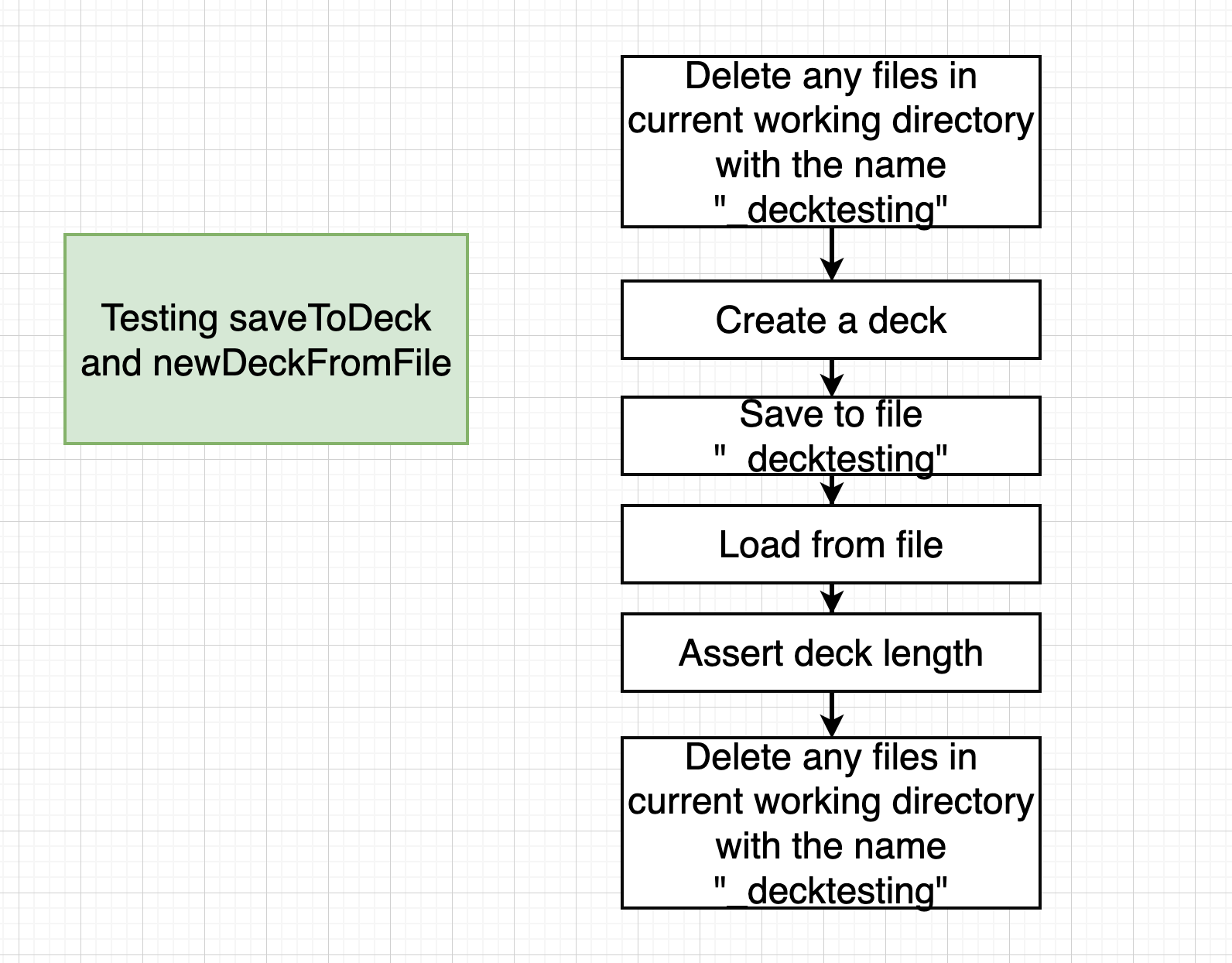


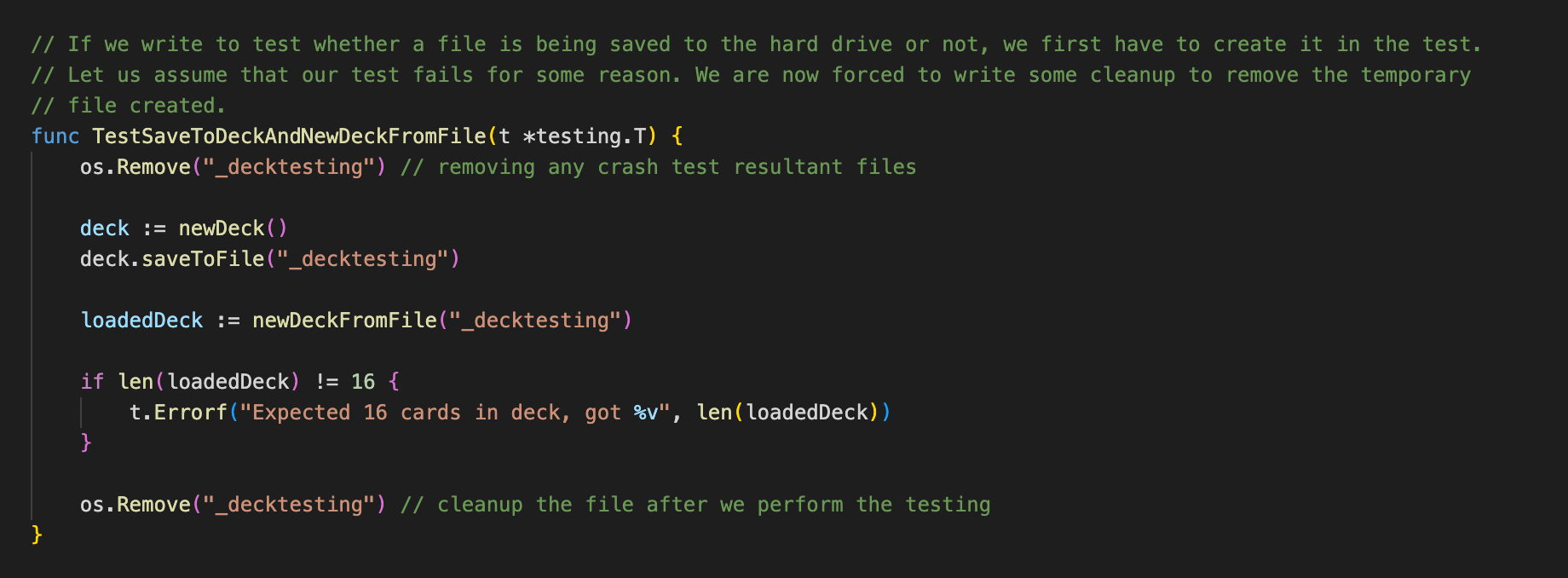
For different functions within a file, we following the above naming convention.

Functions written in a test file start with a capital letter.

\*Testing File IO\*

If we write to test whether a file is being saved to the hard drive or not, we first have to create it in the test. Let us assume that our test fails/crashes for some reason. We are now forced to write some cleanup to remove the temporary file created.

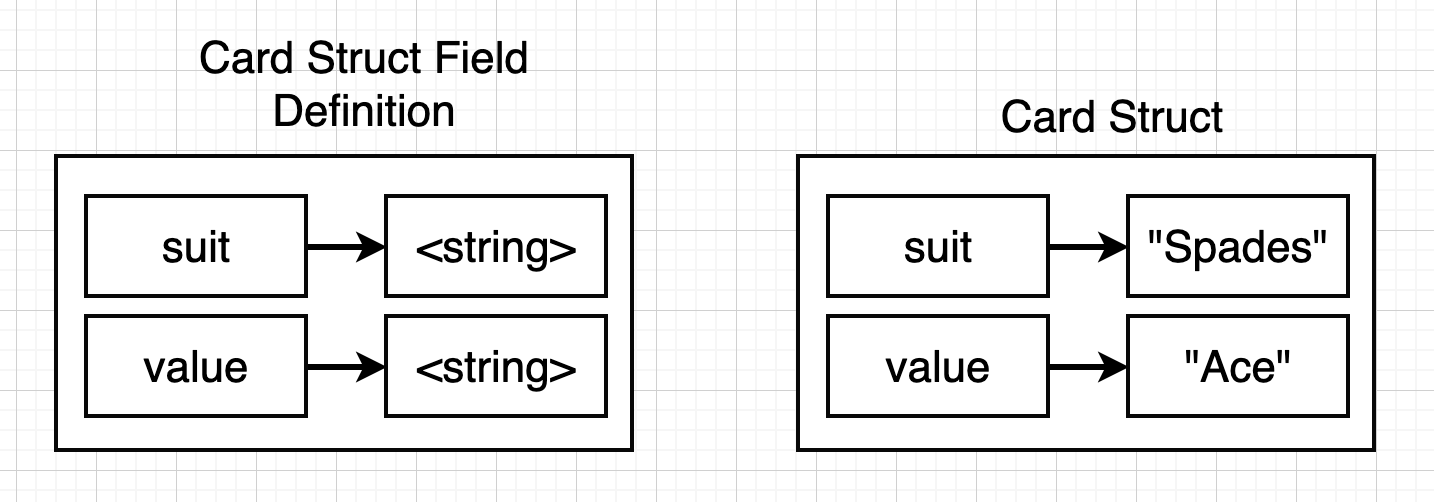




**Section 4**

**Structs in GO**

Collection of properties that are related together.



**// definition of person (of type struct)**

type person struct {

firstName string **// property name**

lastName string

}

func main() {

alex := person{"Alex", "Anderson"} **// Way1: firstName = "Alex" as it follows ordered definition of field names**

alex2 := person{firstName: "Alex", lastName: "Anderson"} **// Way2: propertyName: value**

var alex3 person **// Way3: defining an empty variable will assign an empty struct to it**

fmt.Printf("%+v", alex) **// Printf %+v prints out all property names : values**

fmt.Printf("%+v", alex2)

fmt.Printf("%+v", alex3)

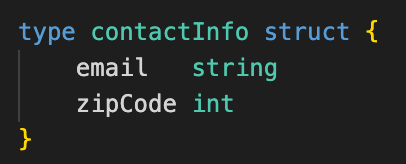
**// updating value in a structure**

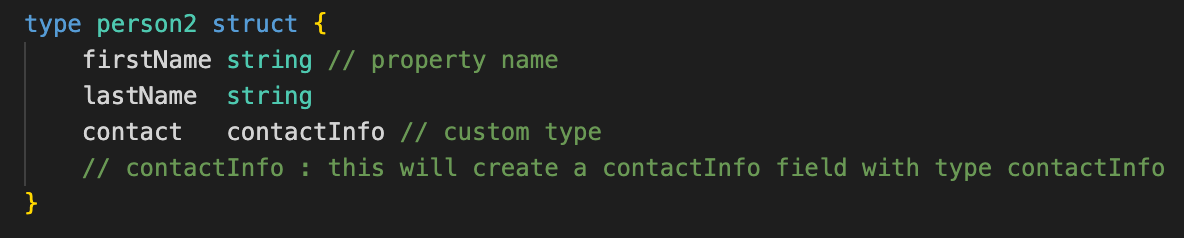
alex3.firstName = "Alex"

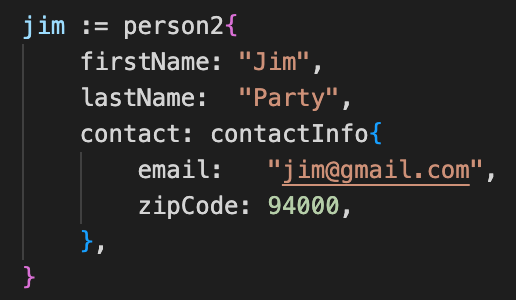
alex3.lastName = "Anderson"

}

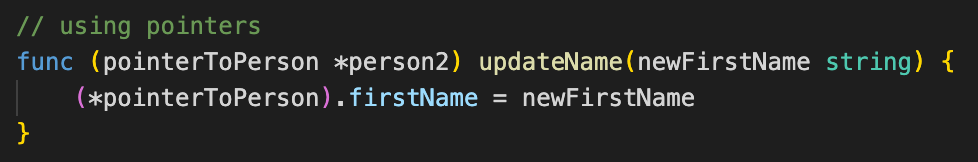
**Embedding Structs**

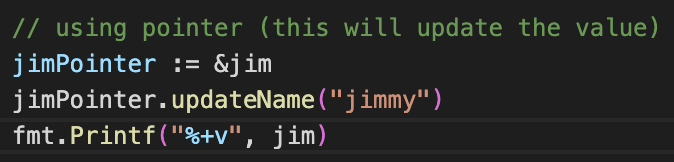






* Go is a pass-by-value language by default.
* We use **Pointers** to make use of pass-by-reference.



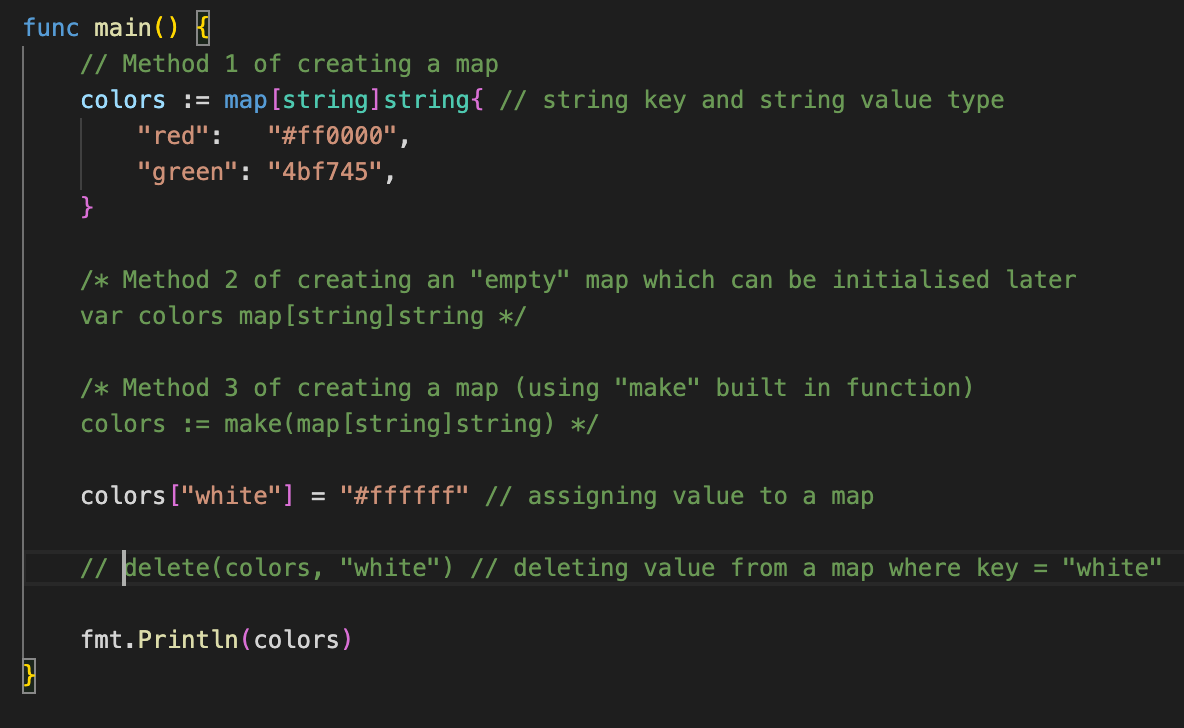


<https://www.udemy.com/course/go-the-complete-developers-guide/learn/lecture/7797348#overview>

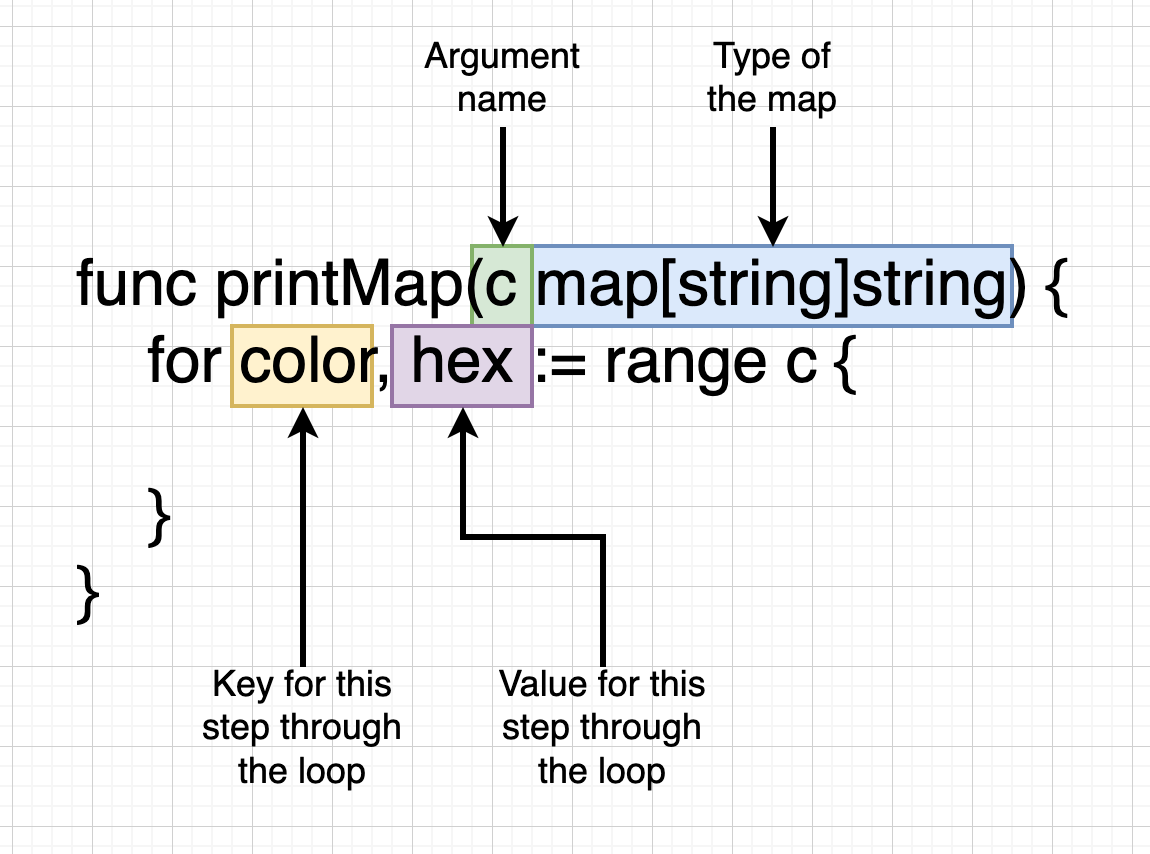
<https://www.udemy.com/course/go-the-complete-developers-guide/learn/lecture/7797354#overview>

**Maps**

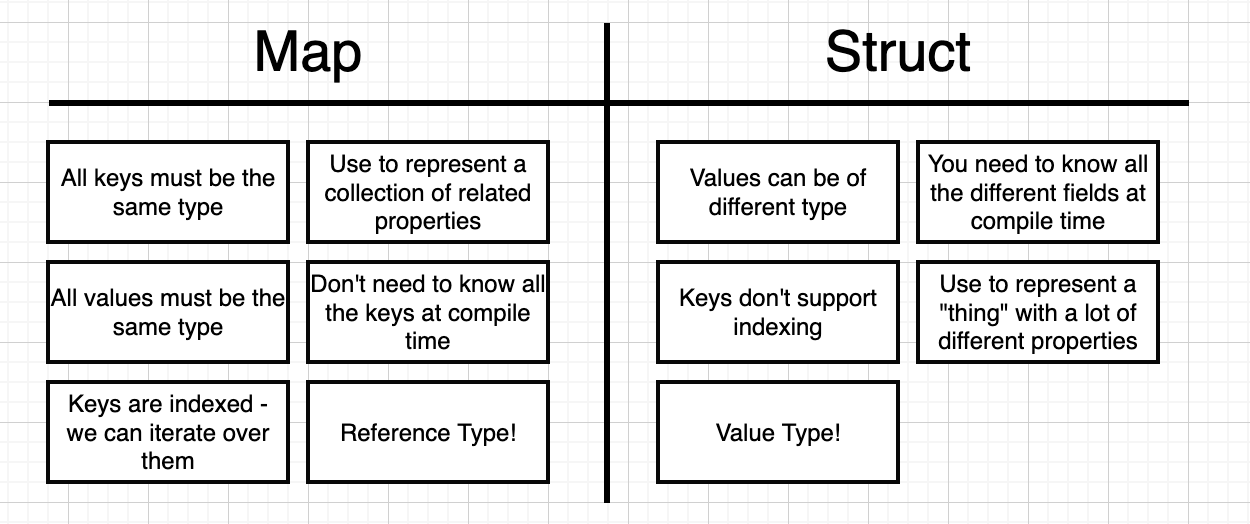
* All keys must be of the same type. All values must be of the same type.
* Ways of declaring maps:



* Iterating over a map.

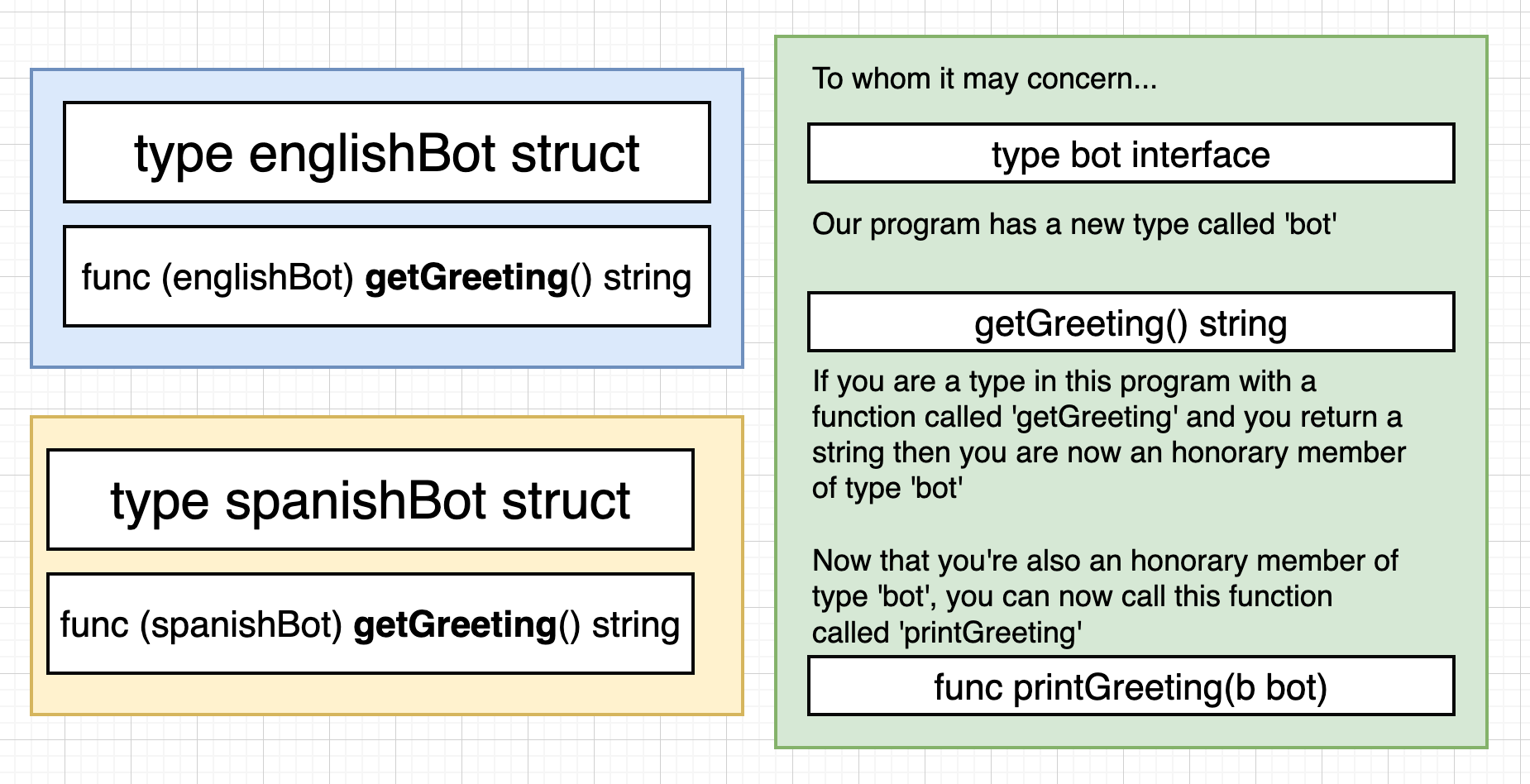


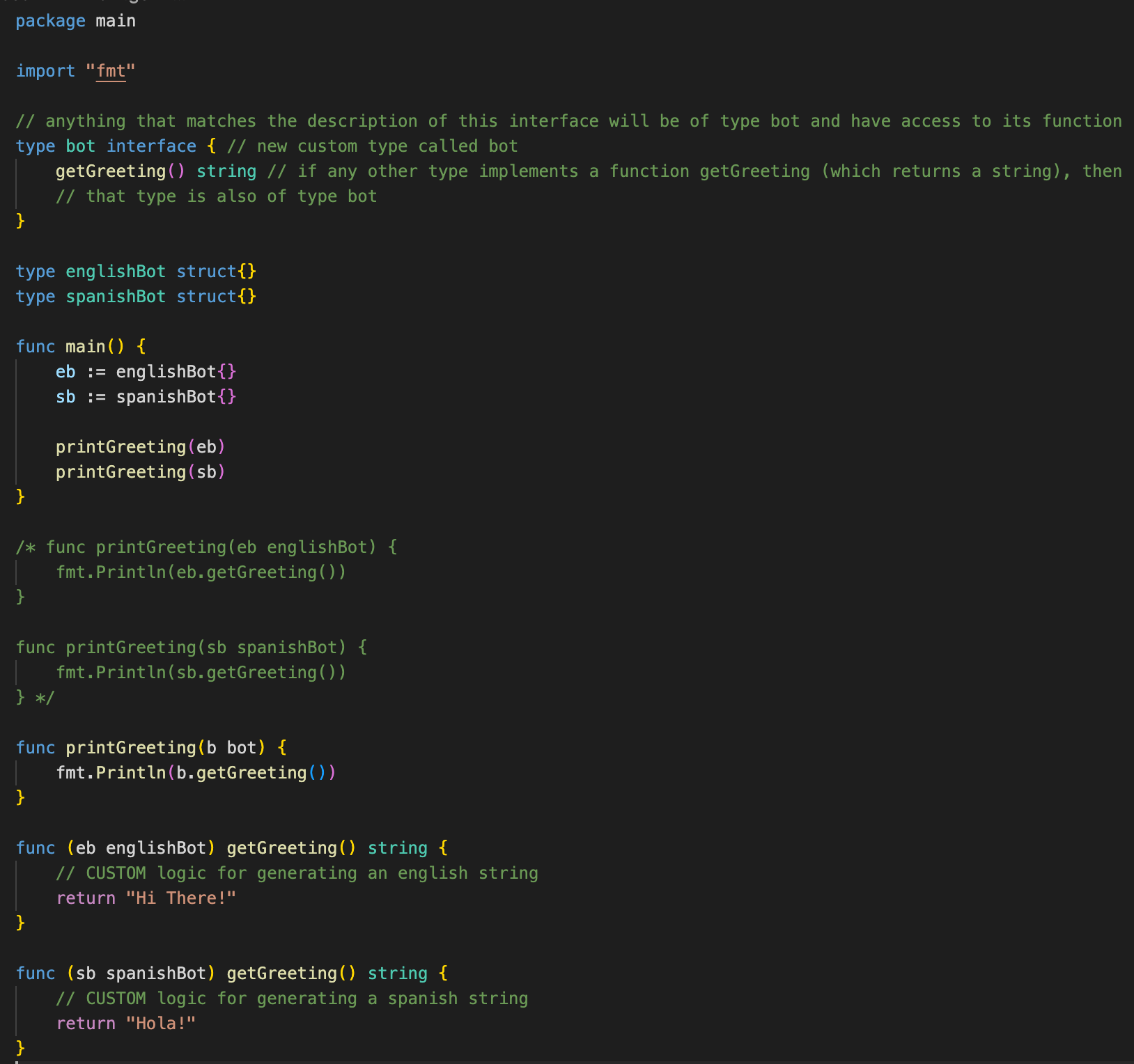
* Map vs Struct



**Interfaces**

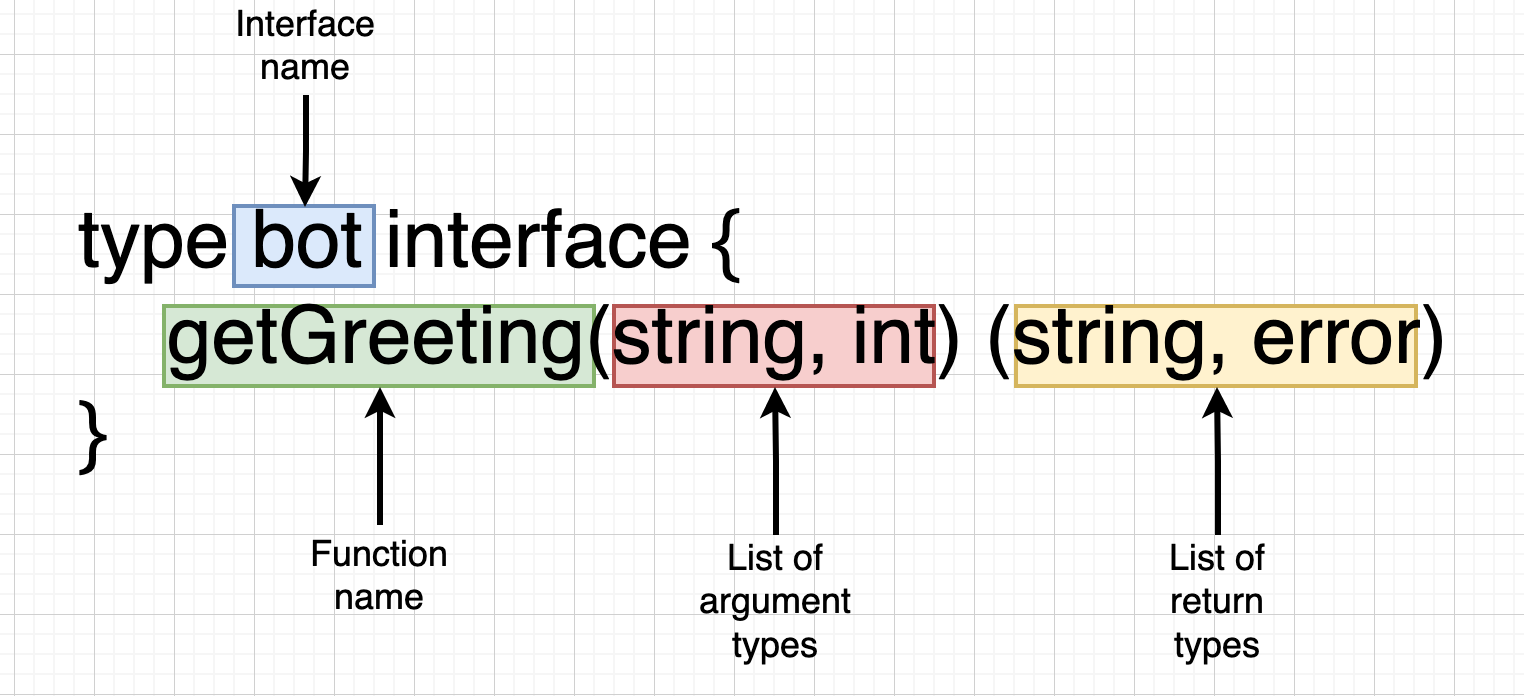
* Since every value has a type and every function must specify the type of arguments it receives, there arises a question about whether we need to re-write functions with the same logic over and over again (just because they accept different data type value). **Interfaces** solve this problem.
* We cannot have function with identical names even if they accept different types of arguments (unlike c++ and java). Their name can be same in case they have different receiver types.
* If a type is implementing the methods defined within an interface then that type is also of type (of the type of) interface specified and can use methods wherein the receiver type is of type of the interface specified.





**Rules of Interface**

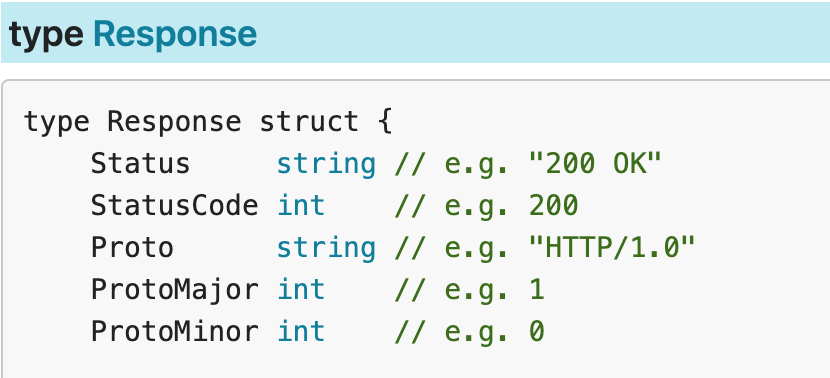
* Any type which also implements all the methods written inside an interface also becomes of type of that particular interface. (a type is said to be implementing a particular method when that function has receiver type as that type)



* We cannot create a value directly out of a type interface (eg bot). We can create value out of concrete types like int, string (built-in type), englishBot (custom-type) etc.
* Interfaces are not generic types (in go atleast).
* Interfaces are implicit, I.e., we don’t manually have to say that our custom type (englishBot) satisfies some (bot) interface. We didn’t say englishBot is also of type bot explicitly.
* If our custom type’s implementation of an interface is broken then interfaces won’t help us (for eg if we return a trash string).

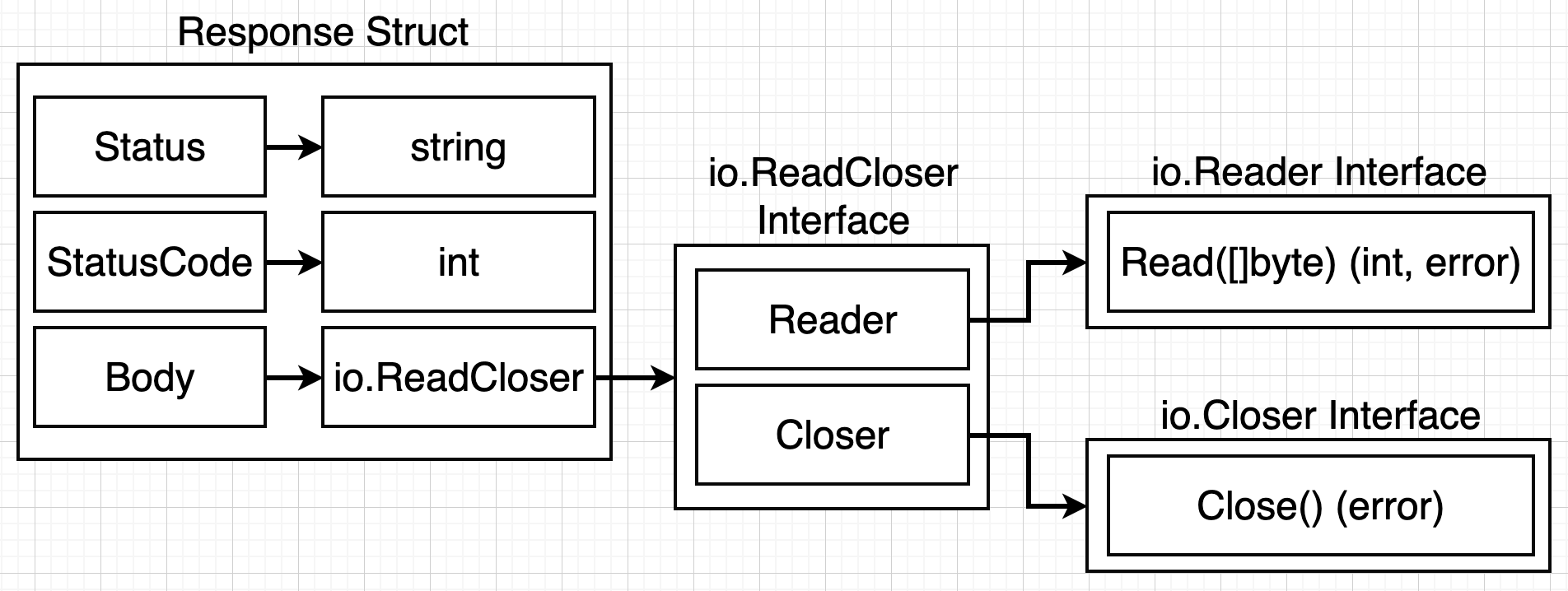
**HTTP Package (inside the NET package)**

1. **func Get(url string) (resp \*Response, err error) :** takes a string as an argument and returns a pointer to a type of repsone (response object) and an error obj.
2. Reponse is of type struct with some of the listed properties and more (like body).



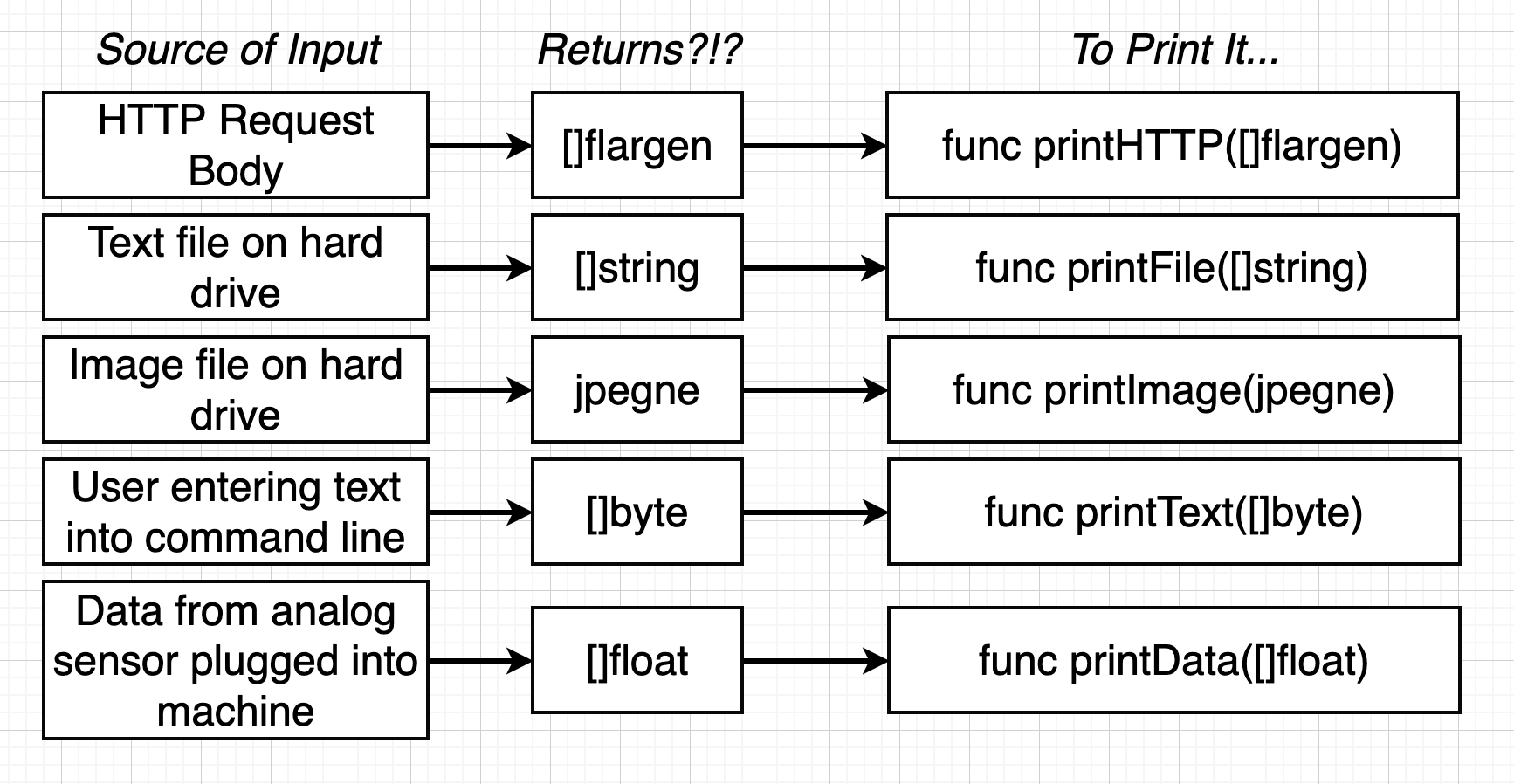
Body is of type **ReadCloser,** an interface, which further implements 2 interfaces (Reader & Writer).

**Body is a field inside a struct which is an interface. Why an interface was used as a type inside a struct? If we specify an interface as a value inside a struct, we are saying that the body field can have any field assigned to it as long as it fulfills the ReadCloser interface. The ReadCloser interface further says that if you want to fulfill the ReadCloser interface (satisfy the requirements) then, you have to satisfy the requirements of the Reader and Closer interface.**

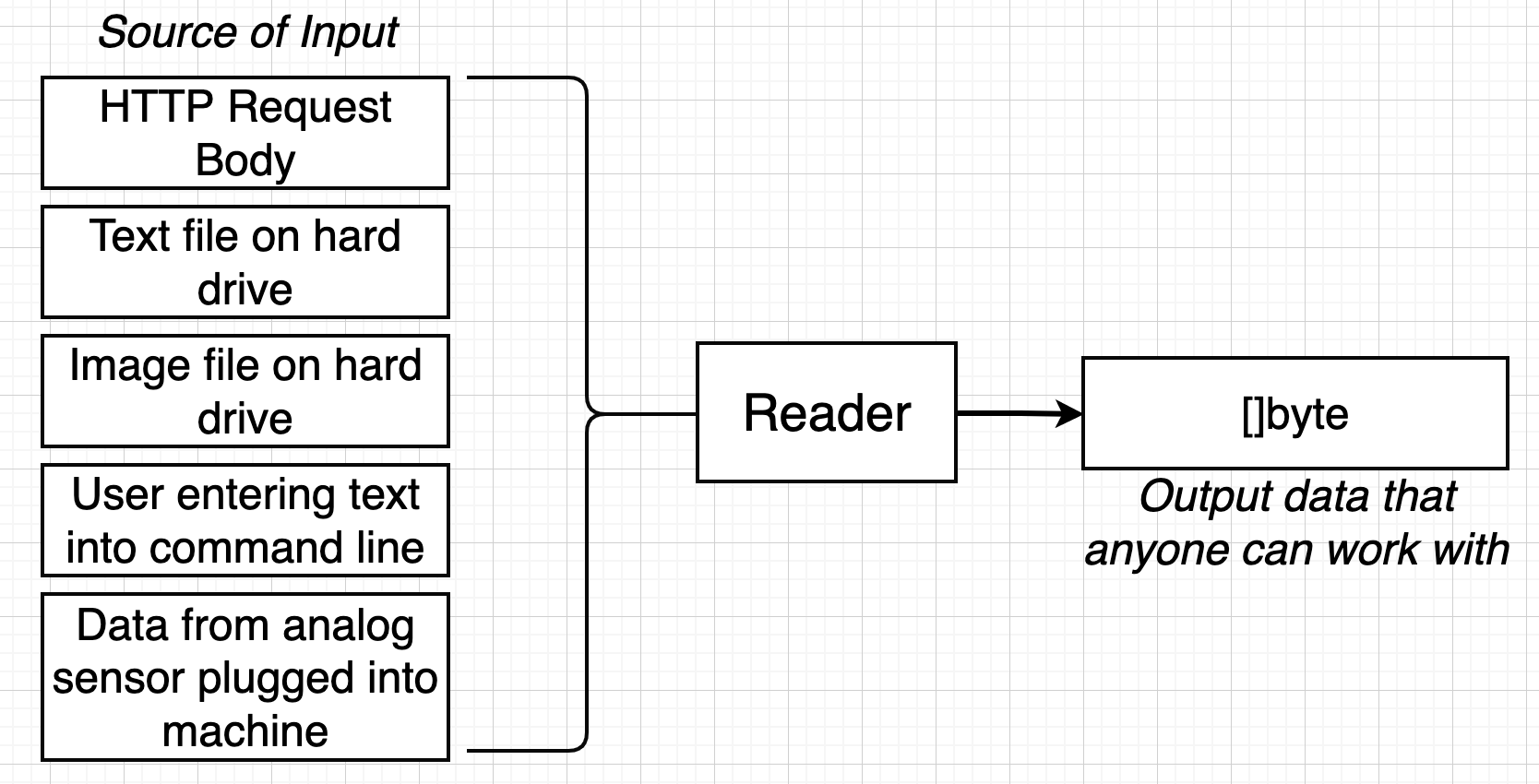


**Reader Interface**

Following is the problem without the reader interface (making multiple class to print data for different types):



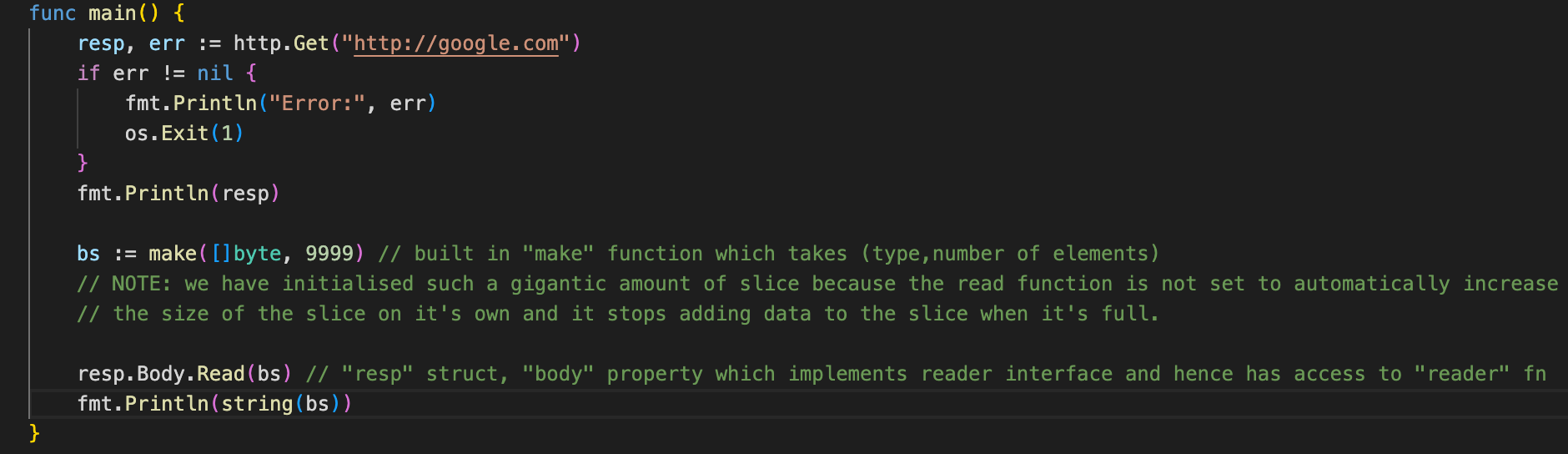
* **The Reader Interface** is an interface/adapter that takes in radically different sources of input and translate them into some common medium that all the other functions we might have can work with.



* Working of the Reader Interface: The thing (eg RequestBody) which wants to consume the data will create it’s own slice (maybe empty), pass it to **Read(p []byte) (n int, err error),** the read function will push data into the slice and then whoever had to call that read function now has some actual data in the slice (pointers/pass by reference in slices like arrays).

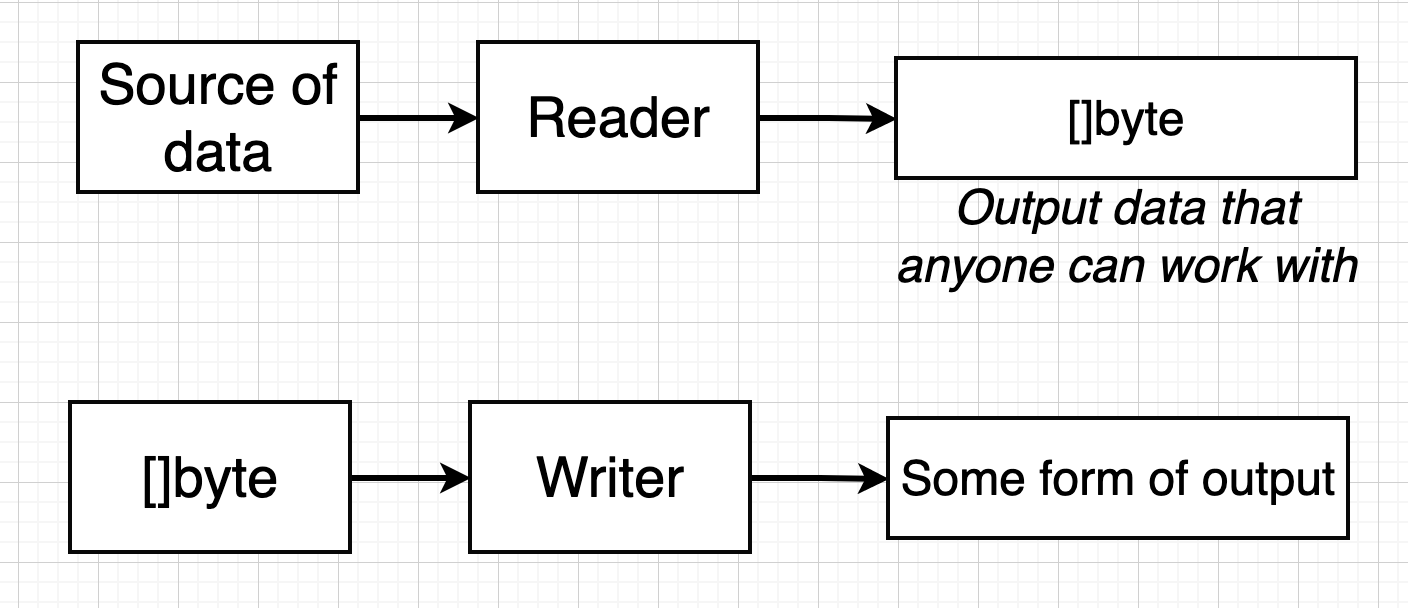


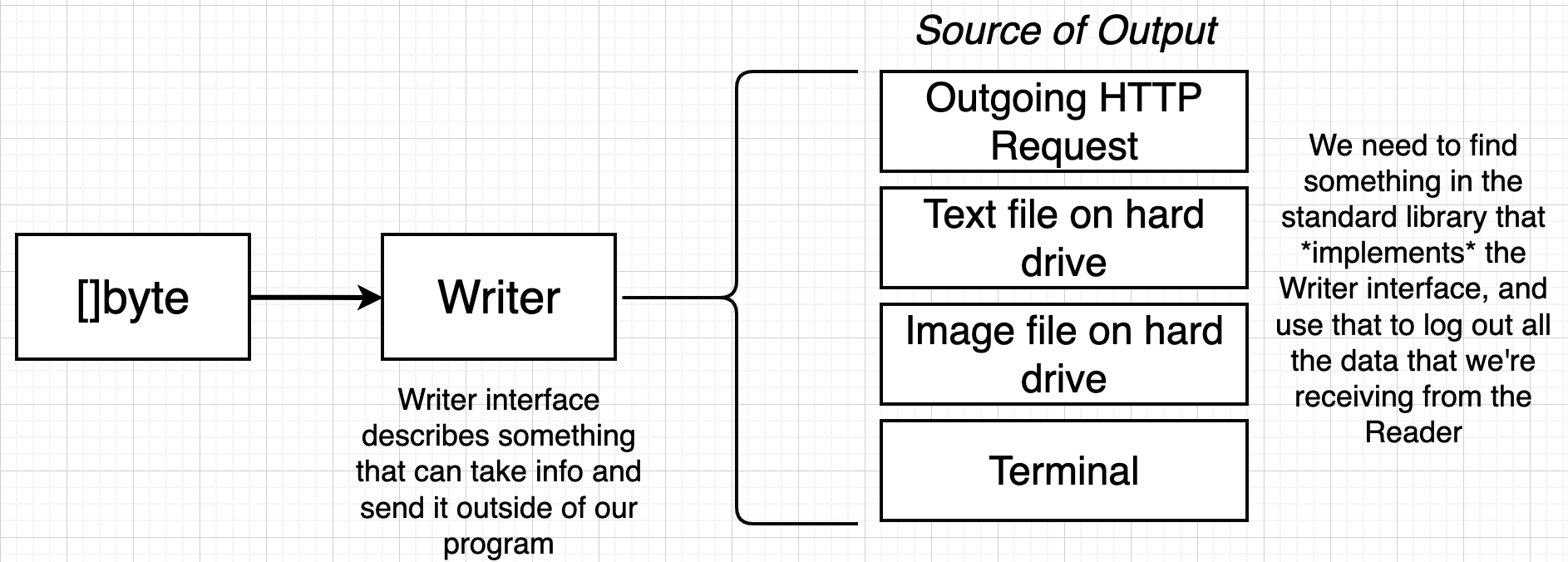
* Writing code that creates an empty byte slice, passes it to the read function (that the responsebody implements) and then see if we can take that byte slice and print the data which was added to it by the read function.



However, we don’t have to always pass an empty slice to the Read function in order to get the byte slice. We can write the following code which does all this work for us (take the responsebody and log out all the information inside of it on our terminal):

**Writer Interface**



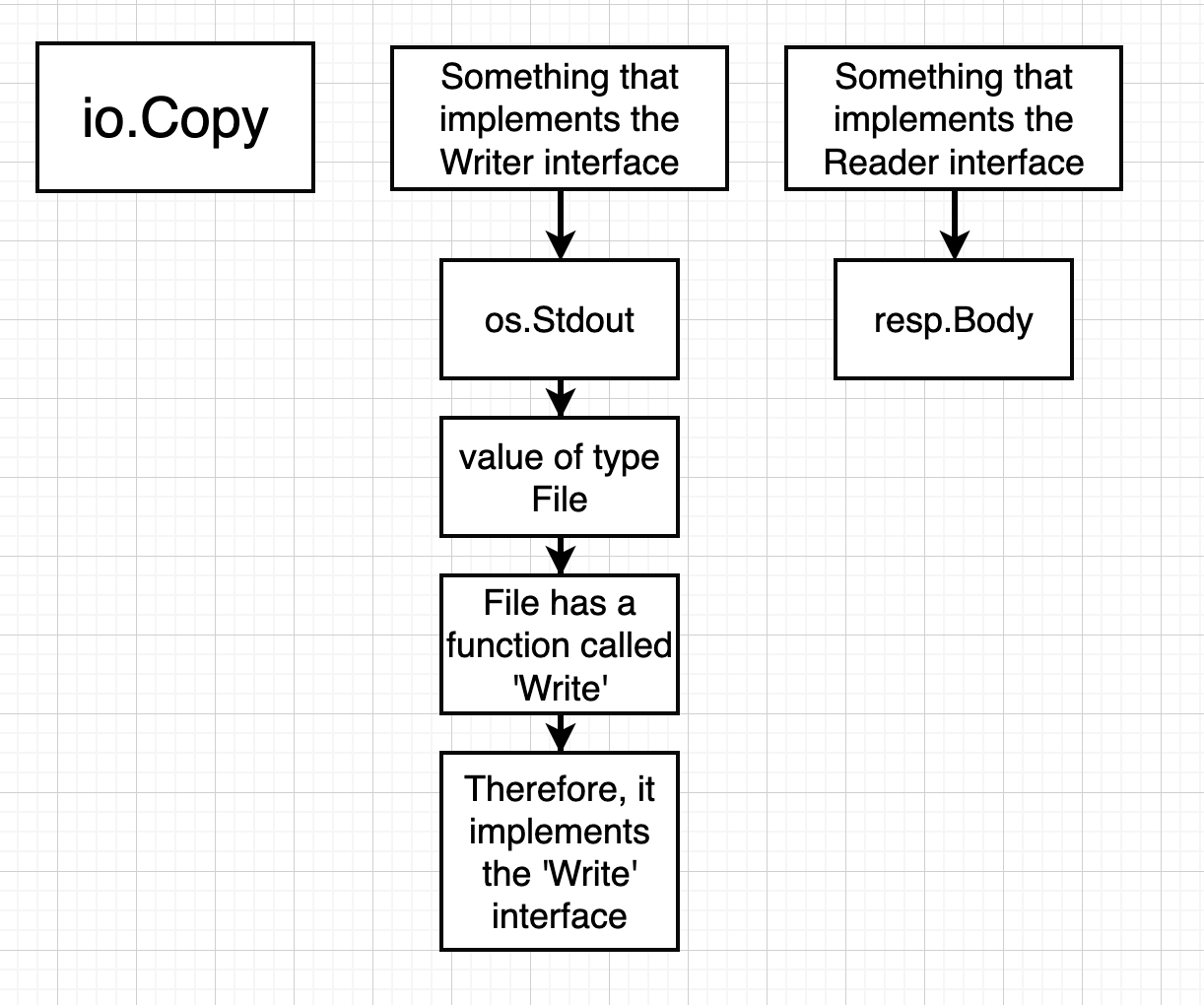


To satisfy the Writer interface, our type must implement a function **Write(p []byte) (n int, err error).** It’s kinda same as the Read function with a minor different that this time it’s actually reading data from the byte slice and then sending data outside (eg to a text file).

We use the following short-hand code for getting the response-body data:

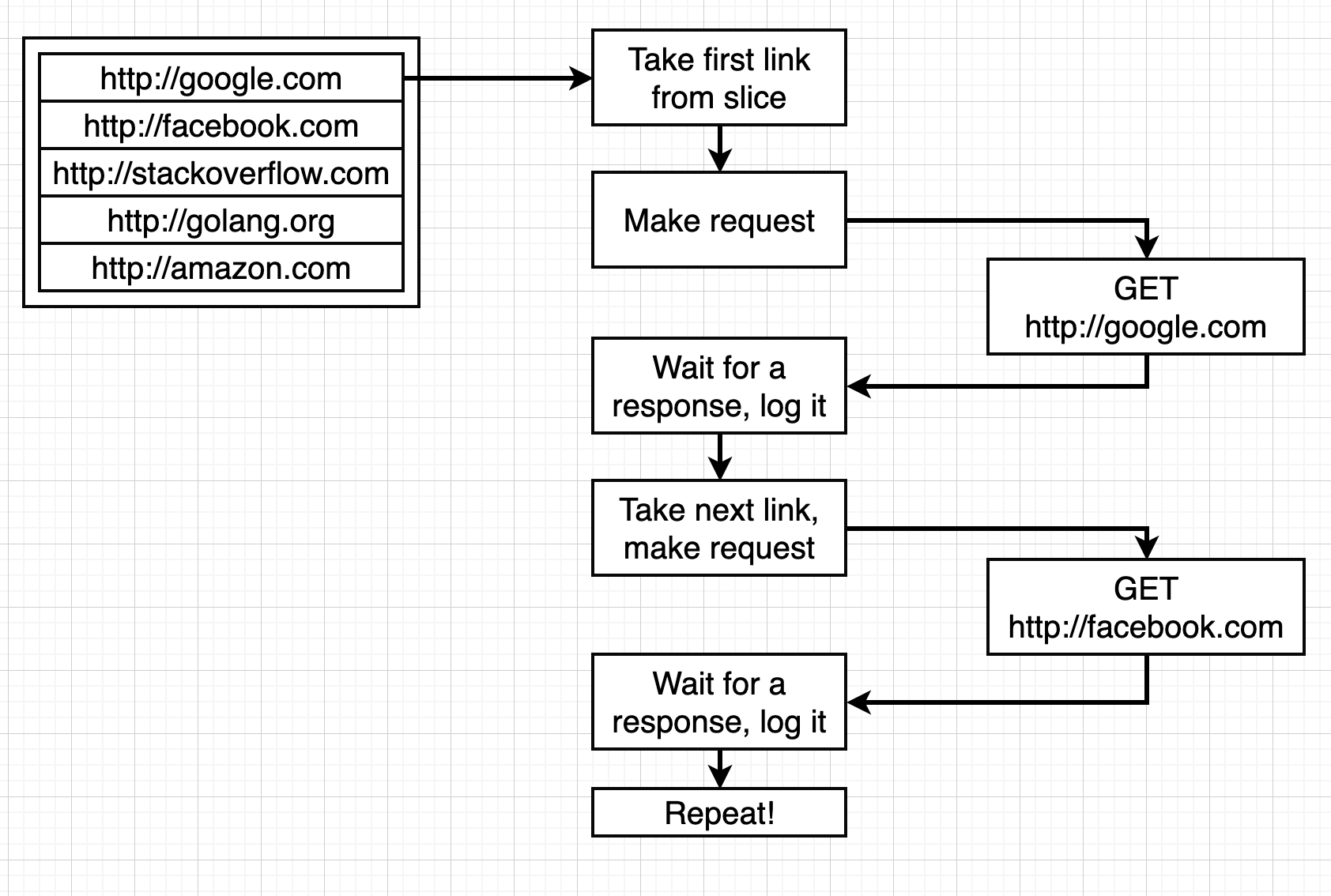
**io.Copy(os.Stdout,resp.Body)**

Following are the 2 arguments:



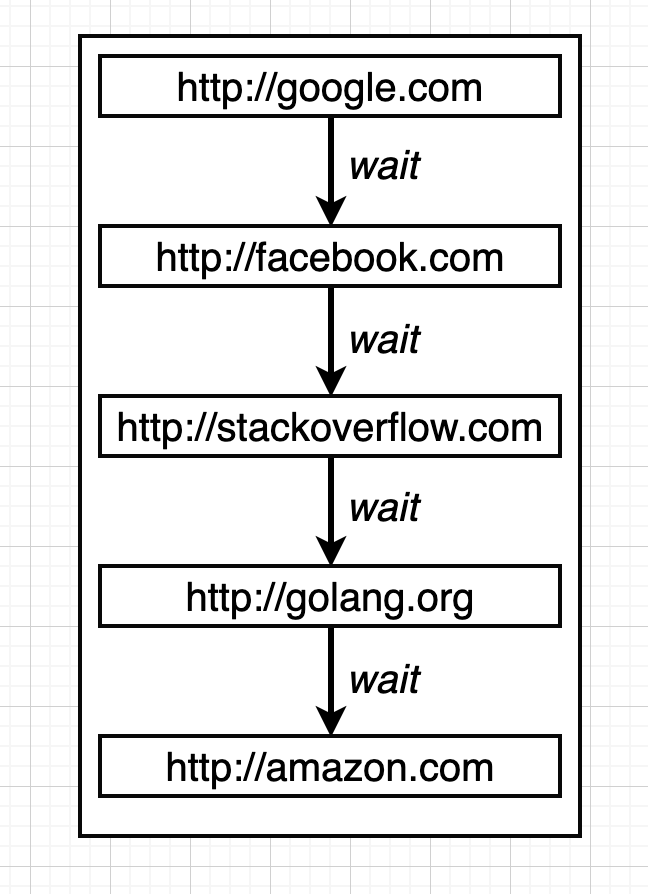
**Channels and GO Routines**

* Used for handling concurrent programming.

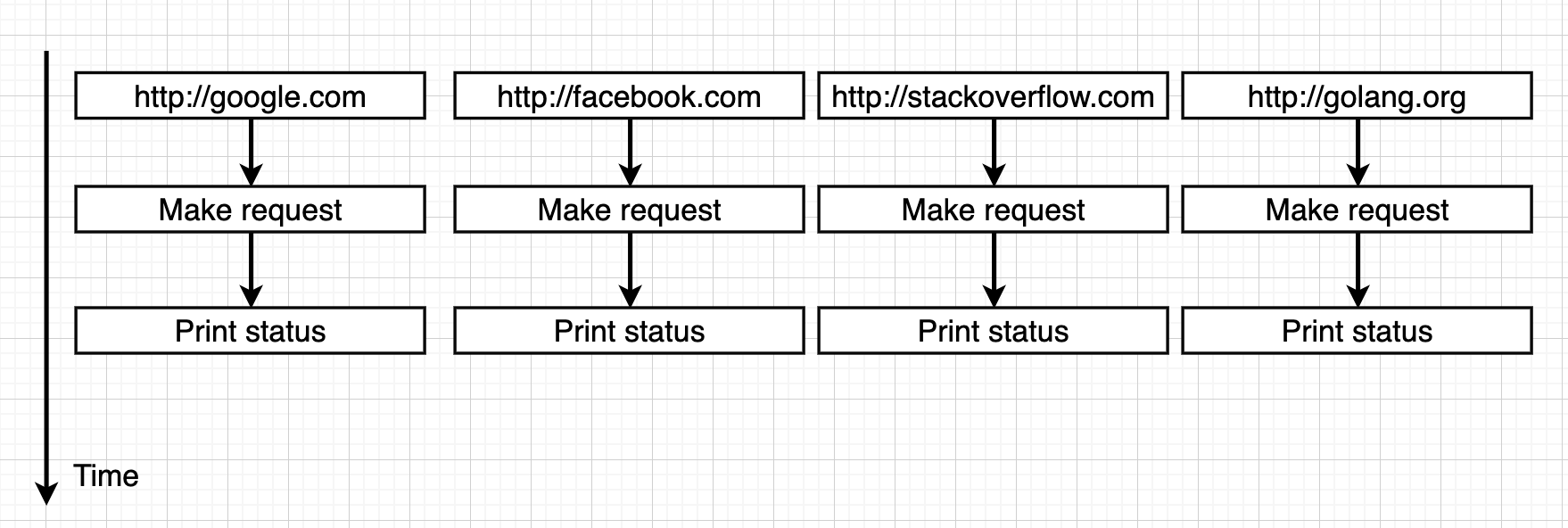


If we use the following serial approach then we can only check 1 link once per let us say an hour (in case there is a huge list of links) since we have to first completely iterate the list to reach the top again. Also, when we are waiting for 1 request, our cpu is idle.





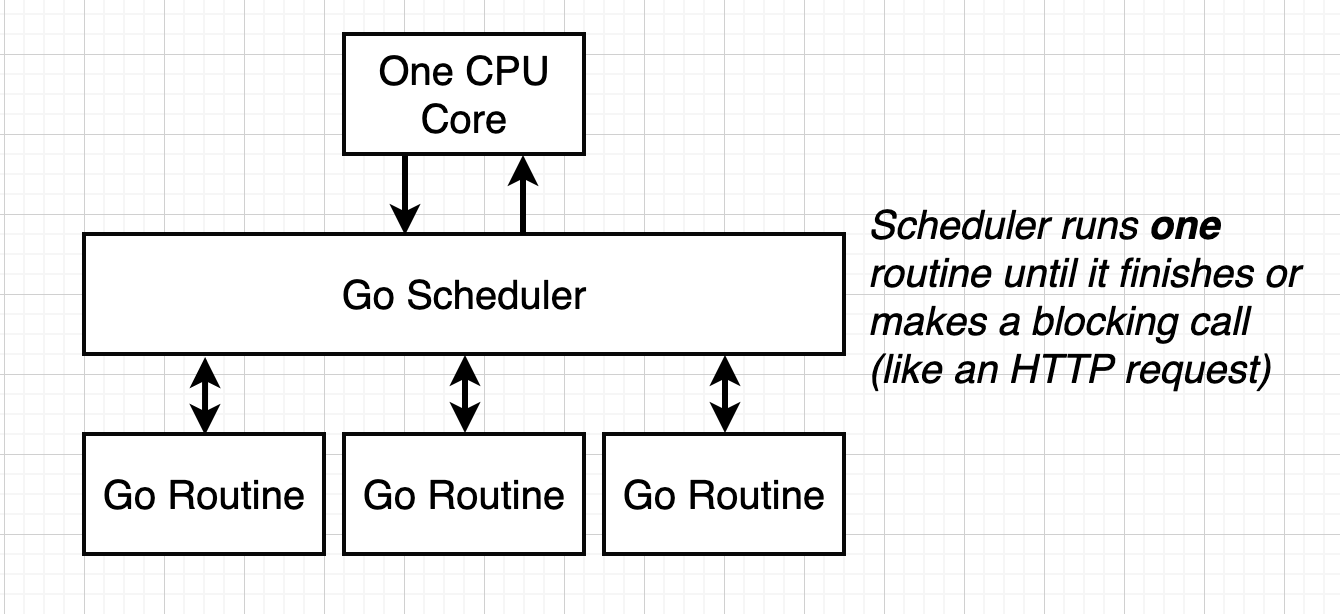
Hence, it's better to use the parallel approach. Here we fetch multiple requests at the same time (using **Go Routine)**.



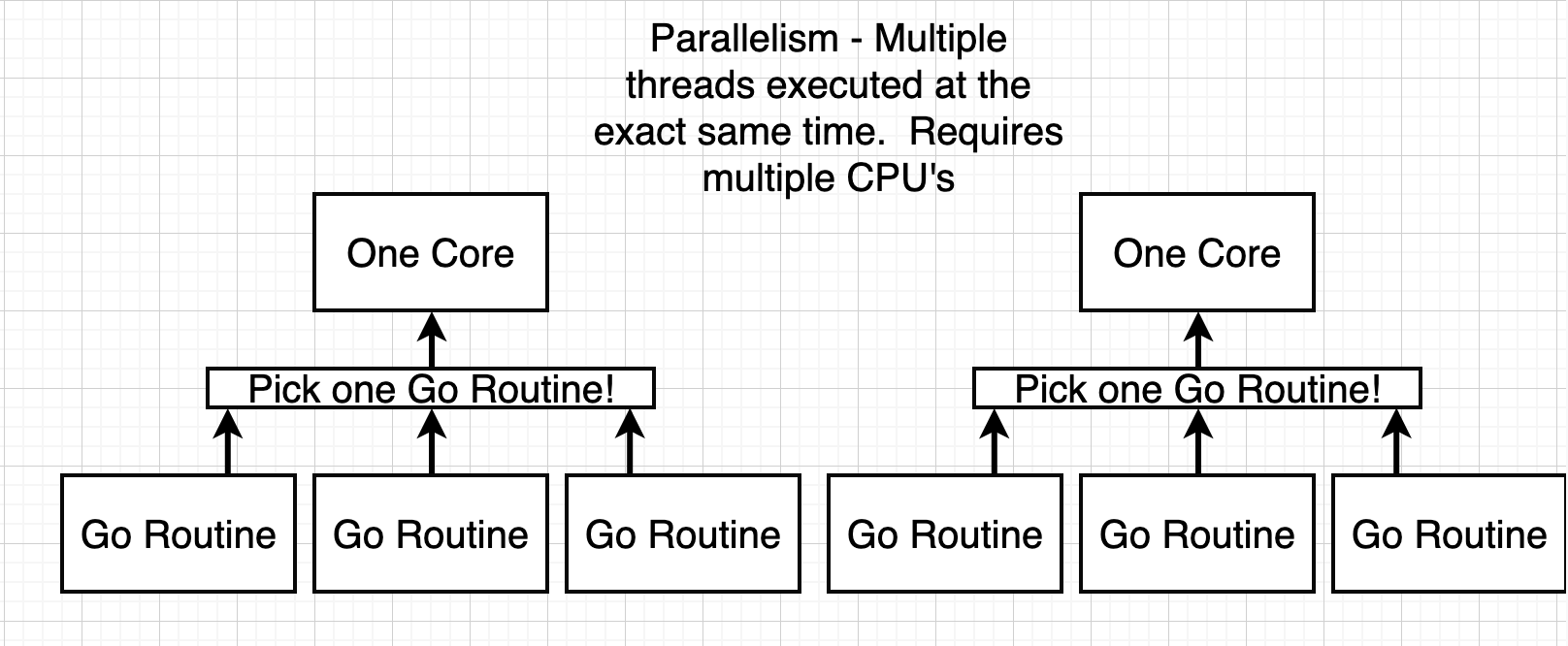
A **GO Routine** basically runs our code line-by-line. A single go routine is automatically created when we run our main.go file. We can create multiple go routines by specifying the go keyword in front of the function call.

**By default GO Routines use a single core and hence we are using concurrency by default and hence, only a single go routine is running at a particular instant of time.**

**Concurrency:** we can keep running multiple go-routines without any break (like waiting for a page to send results) but not kind of at the same time. Only 1 go routine is being executed at once until it’s task is completed or we have to wait for a response or something.



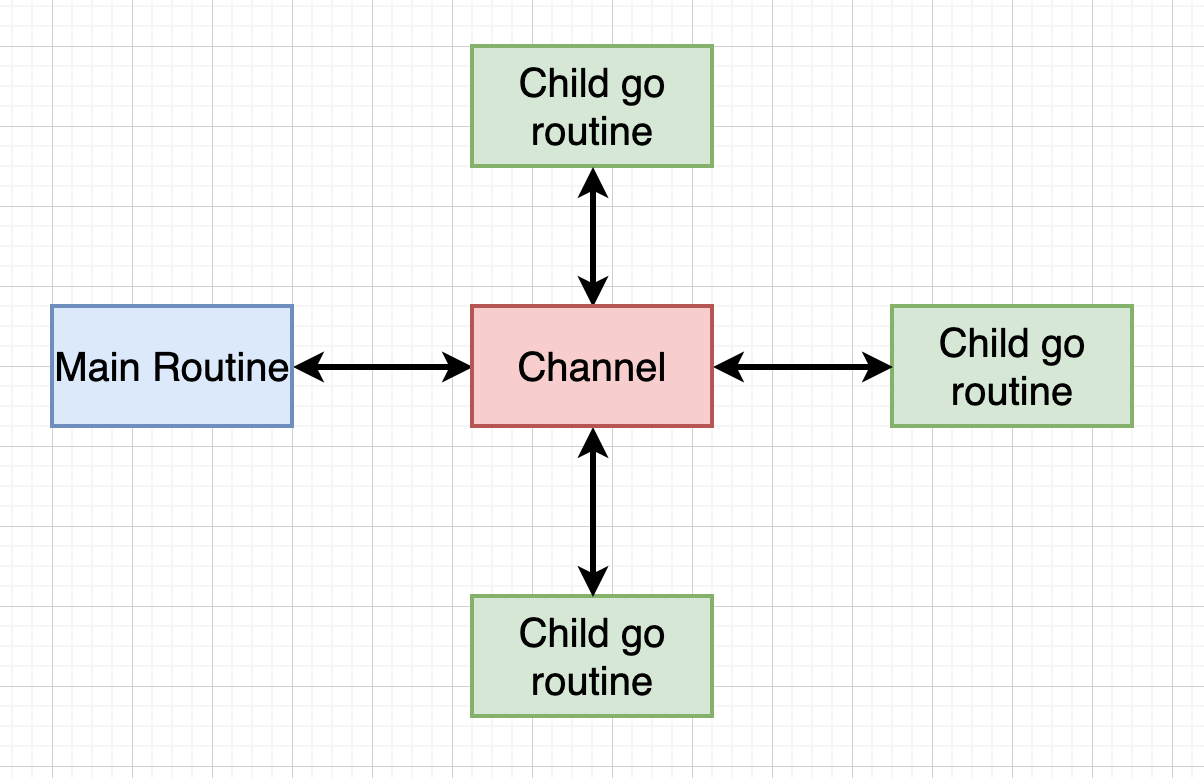
**Parallelism:** since there are multiple cores, we are able to do multiple things at the same time (like running multiple go routines at the same exact time).



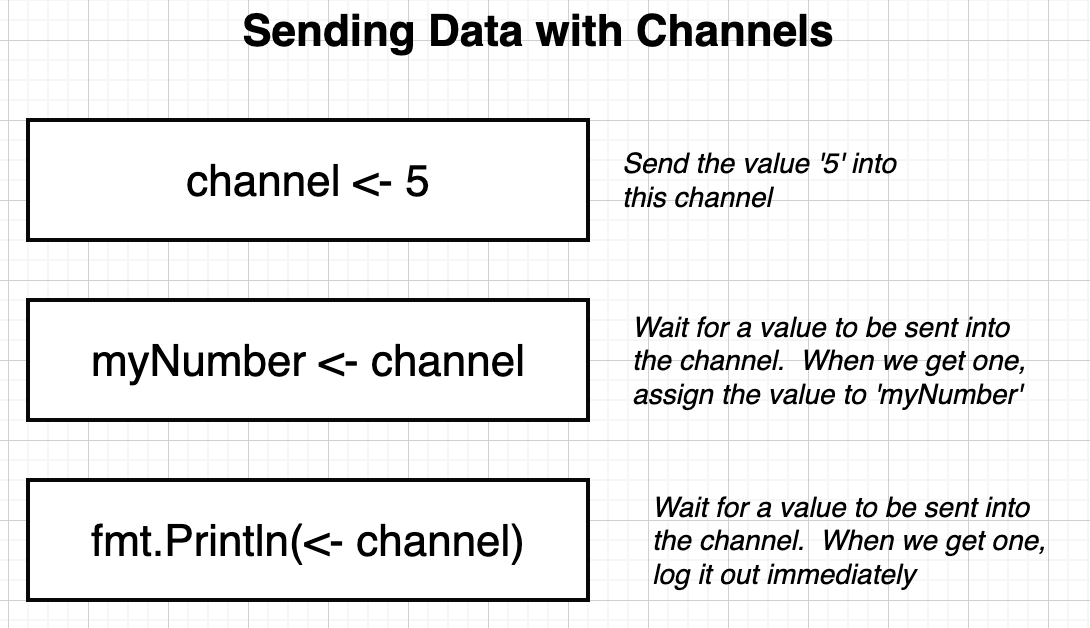
However, the main go routine (created by default) is the one which controls the execution and stopping of the program. The child go routines (created by us) have less importance than the main go routine. As soon as the main go routine is done finishing its work, the program is terminated even if the child go routine has not finished it’s work and hence there arises a need to use **channels.**

**Channels**

Channels are used to communicate between different go routines so that the main go routine can communicate and know when the child go routine has completed it’s work.

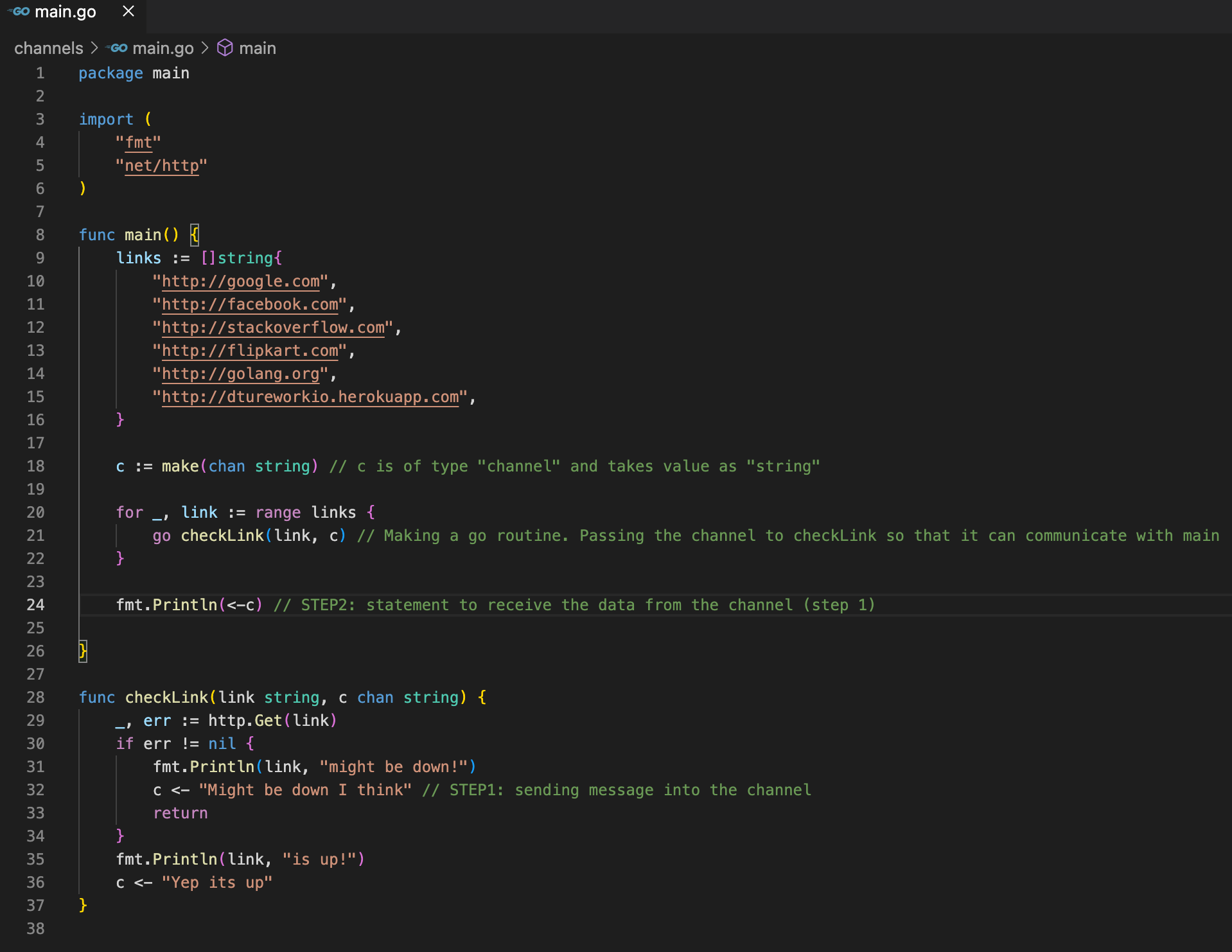


* Channels are of a particular type (eg string, int etc) and we can communicate between different go routines by passing real values of the particular channel type.



Code to make a channel (in main): **c := make(chan string)** c is a value of type channel and takes string.

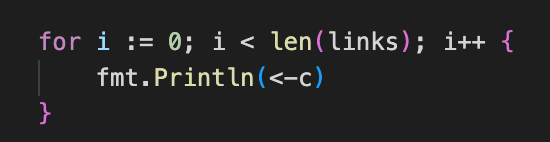
***\*Receiving value from a channel is a blocking line of code (fmt.Println(<-c) in main)\****



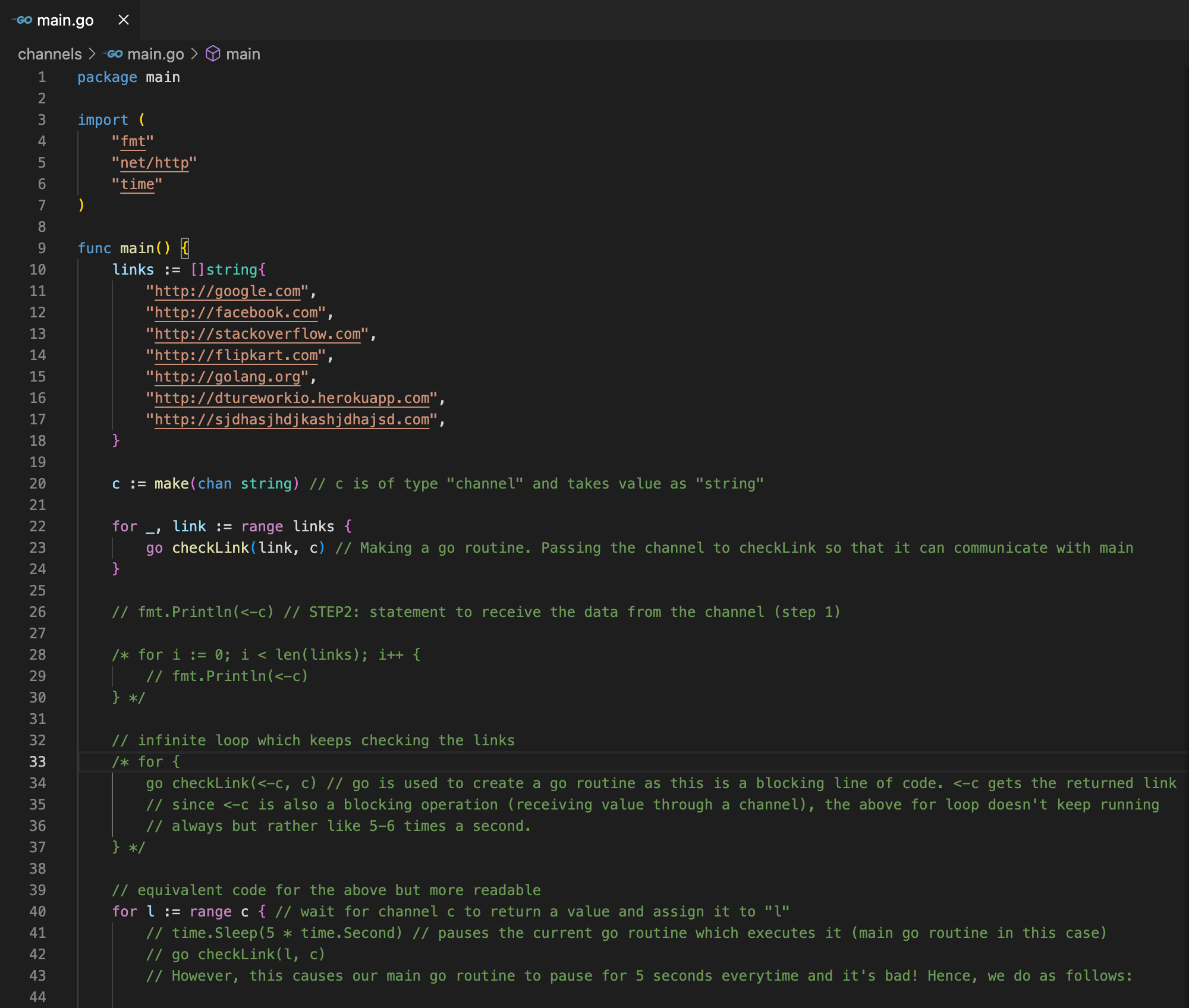
In the above code, only 1 of the link is returned and the reason for this is as follows:

Receiving value from a channel is a blocking line of code (line 24). Hence, after executing line 21 5 times, our main go routine is waiting for a value to be sent from the channel and as soon as any of the 5 child go routine send the data, it is printed and the program exits.

To solve the above problem, we write a for loop to get the results.



Final Code





<https://www.udemy.com/course/go-the-complete-developers-guide/learn/lecture/7824514#overview>