course repo: <https://github.com/StephenGrider/GoCasts>

The diagrams shown in the course are attached to this lecture note as a zip file.

Download the file and extract it somewhere on your computer.

Visit diagrams.net (formerly draw.io).

Select Open Existing Diagram and use the file explorer to select the diagram file from your computer.

or

Click on File from the diagrams.net menu.

Select Open From Device and use the file explorer to select the diagram file from your computer.

Main commands used in Go:

go build -> compiles source code into a binary

go run -> compiles and runs source code

go fmt -> formats 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 project

A package is a collection of common source files

The only requirement for every file inside of a package is that the first line of the file must declare the package that it belongs to.

There are two types of packages:

Executable – generates a file that we can run

Reusable – code used as ‘helpers’, good place to put reusable logic (code dependencies, libraries)

How do you know which type of package that you’re making?

The word ‘main’ is used to make an executable type package.

Any other name is for making a reusable type package.

Every executable packaged (named main) must also have a ‘main’ function where execution begins.

standard packages in go:

https://pkg.go.dev/std

Go has two types that include lists of variables:

1. Array – fixed length list
2. Slice – an array that can grow or shrink

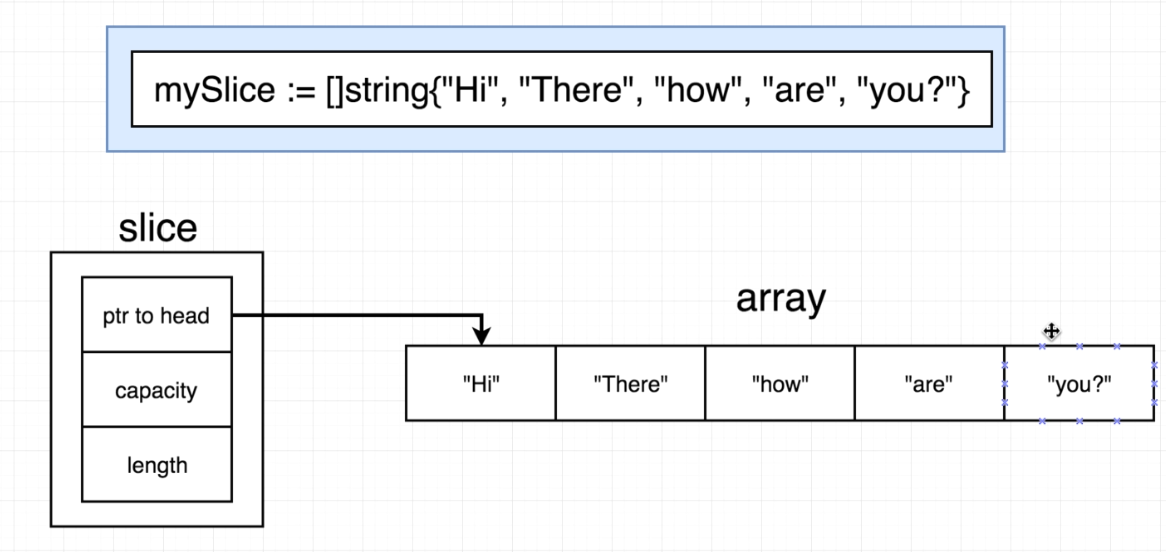
Every element in a slice must be of the same type.

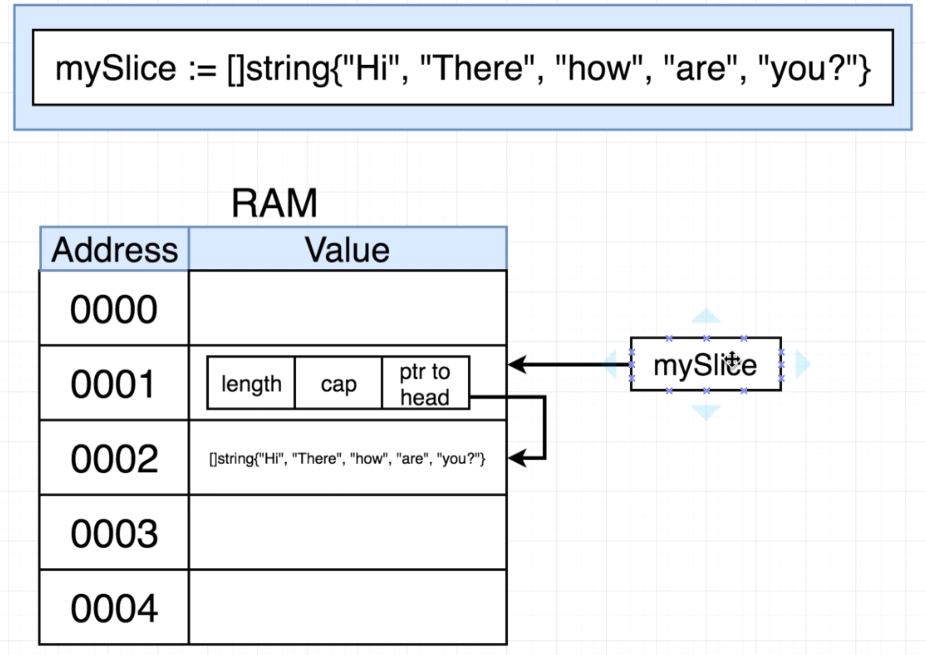
Slices vs Arrays

A slice is another data structure with pointer, capacity, and length, but it is built on top of an array.

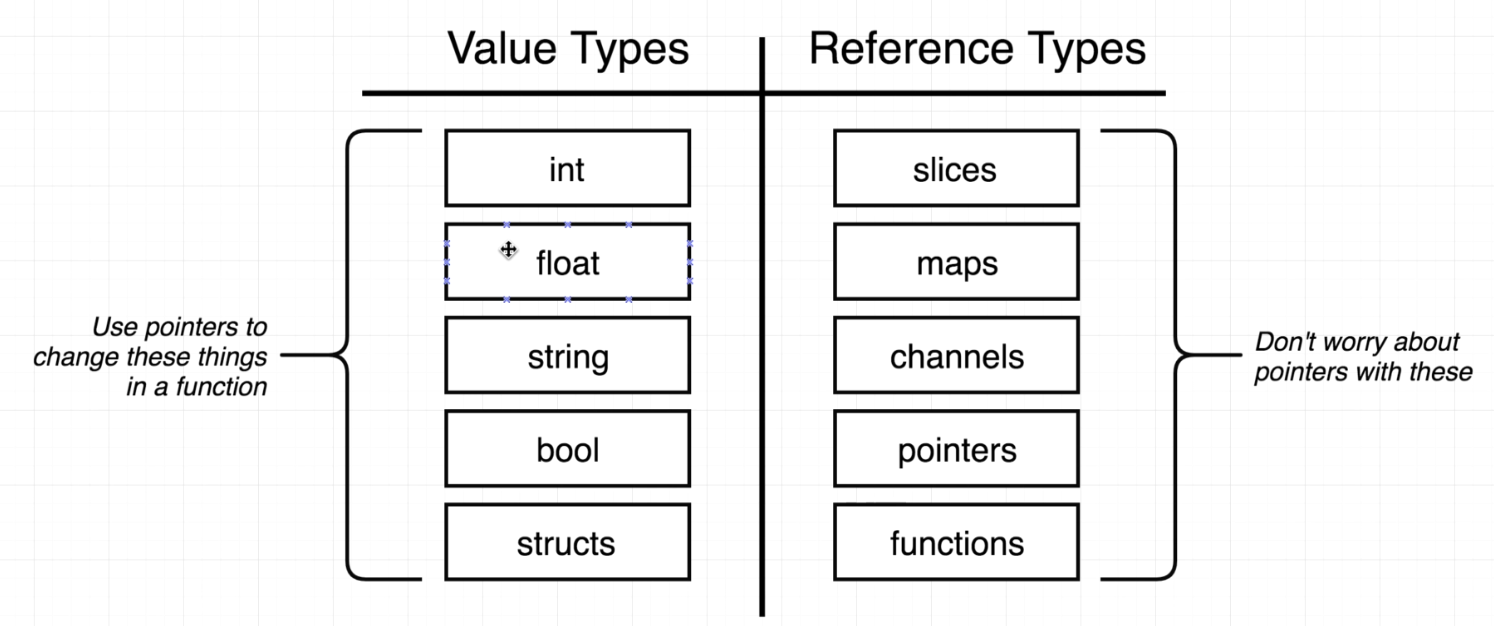
Since a slice is a pointer, even though it is passed by value, the pointer still points at the array, and modifications to the underlying array will persist.

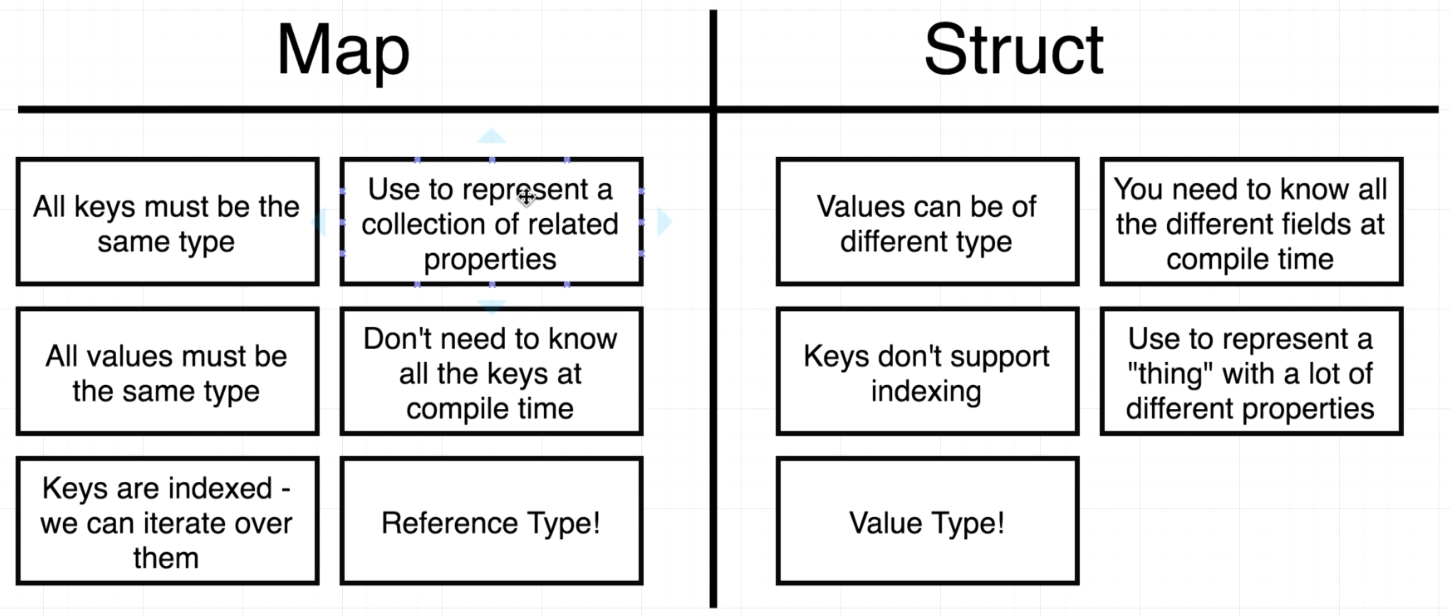
Because of this, we would call the slice a “reference” type.





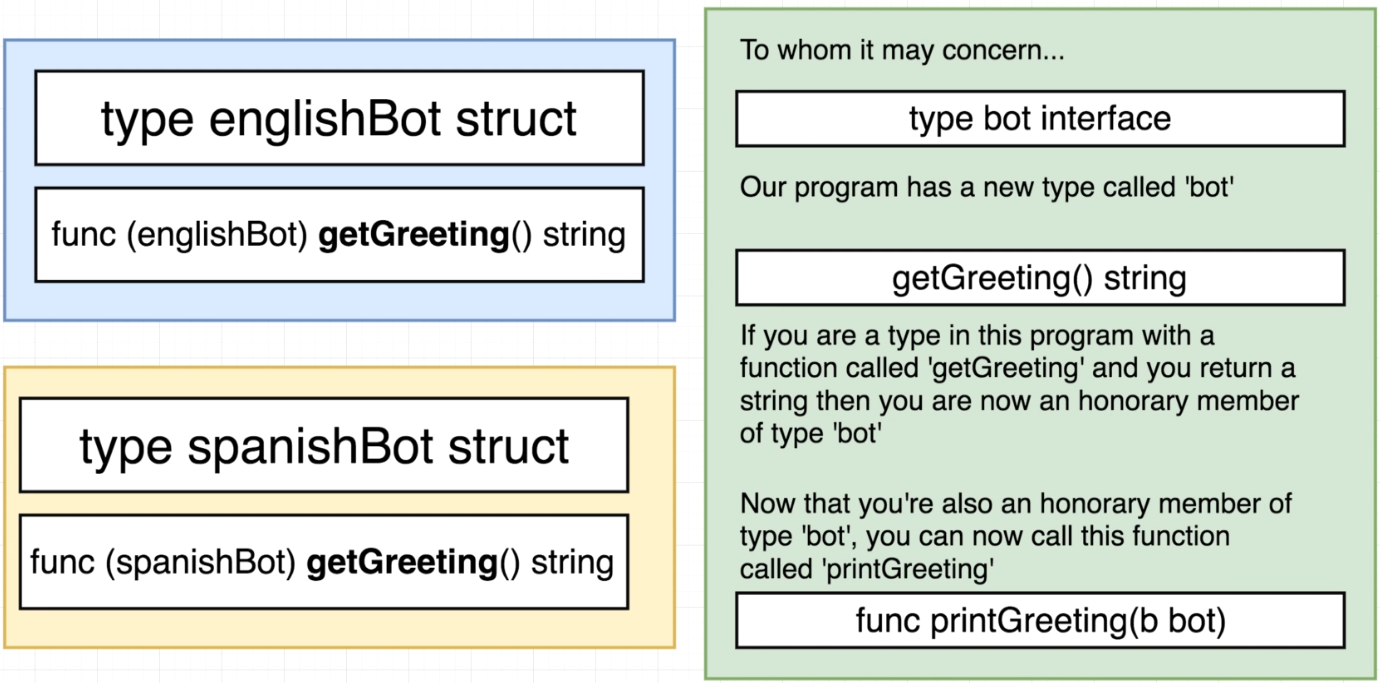
There are several other reference types:

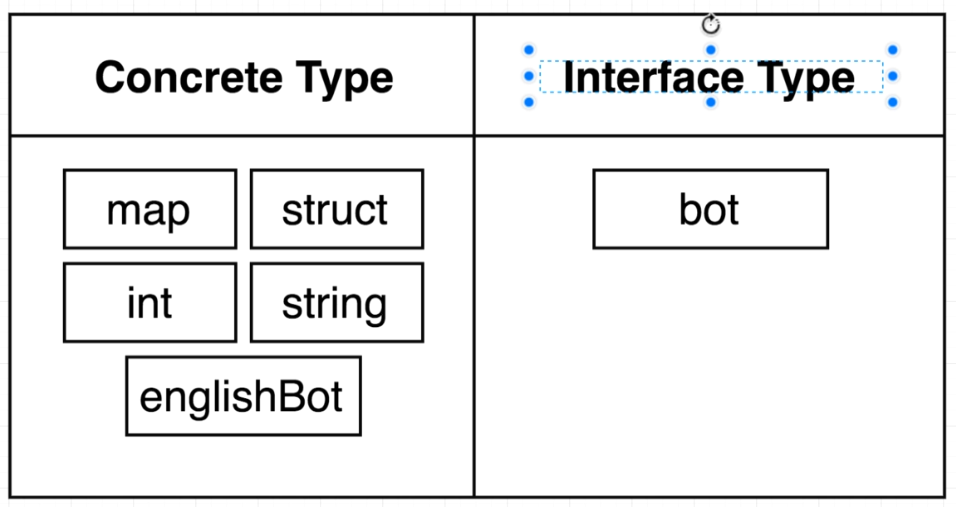


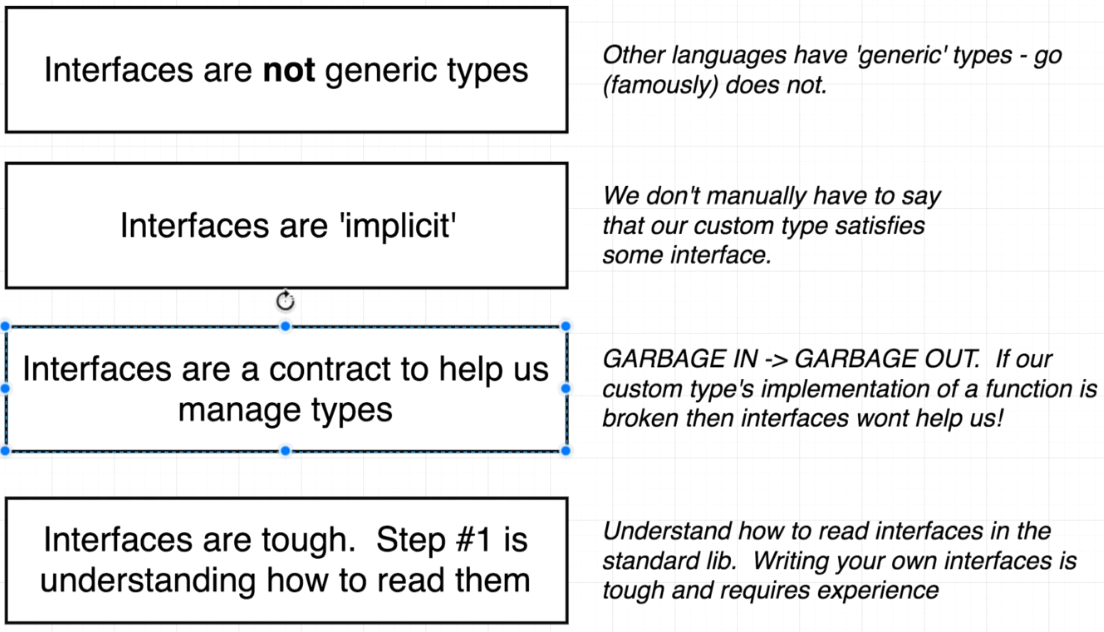


Interfaces

One of the things that interfaces is trying to solve for us is re-usability of code. For example, we wrote the shuffle() function for the deck type in our first program. If we wanted to shuffle another type, e.g. a slice of floats, a slice of strings, etc., we can use an interface.

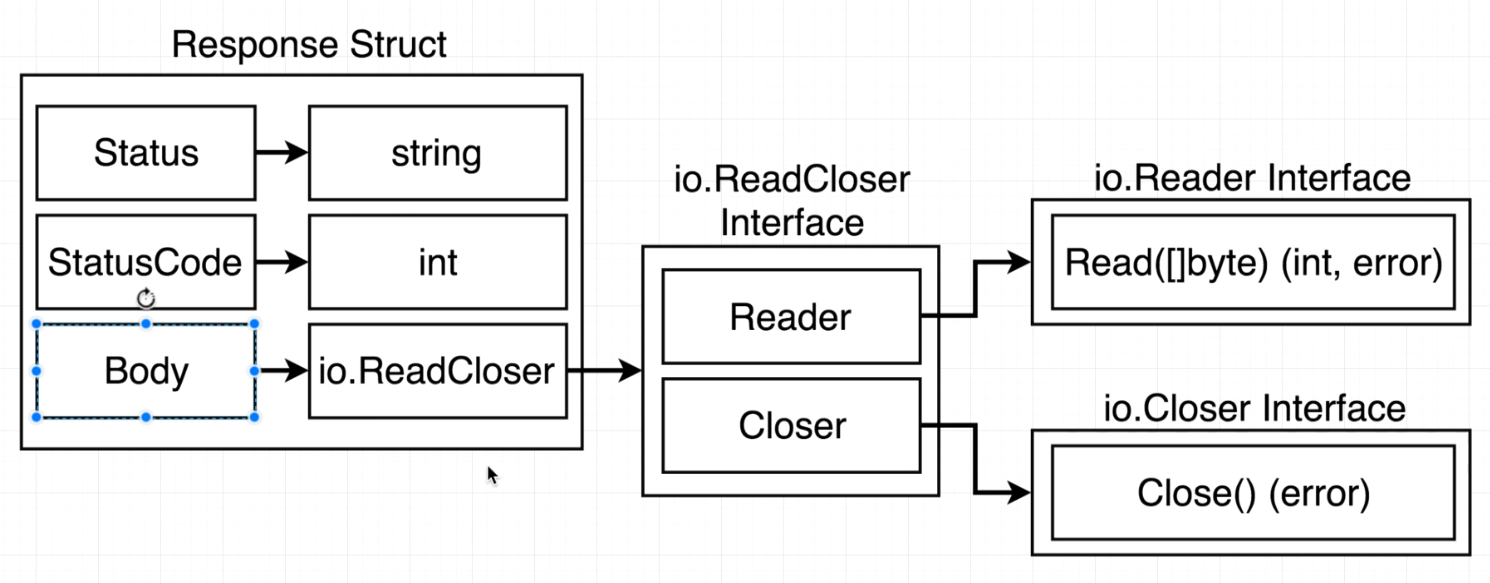






Correction: Go does now support generics

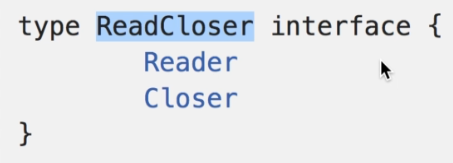
If we specify an interface type inside of a struct, then we can give it any type for a value as long as it satisfies the interface.



In the case of the Response struct (found in the net/http package), we can create our own struct that satisfies the io.ReadCloser interface. As long as we implement a Read() and a Close() in the struct, it will satisfy the io.ReadCloser type.

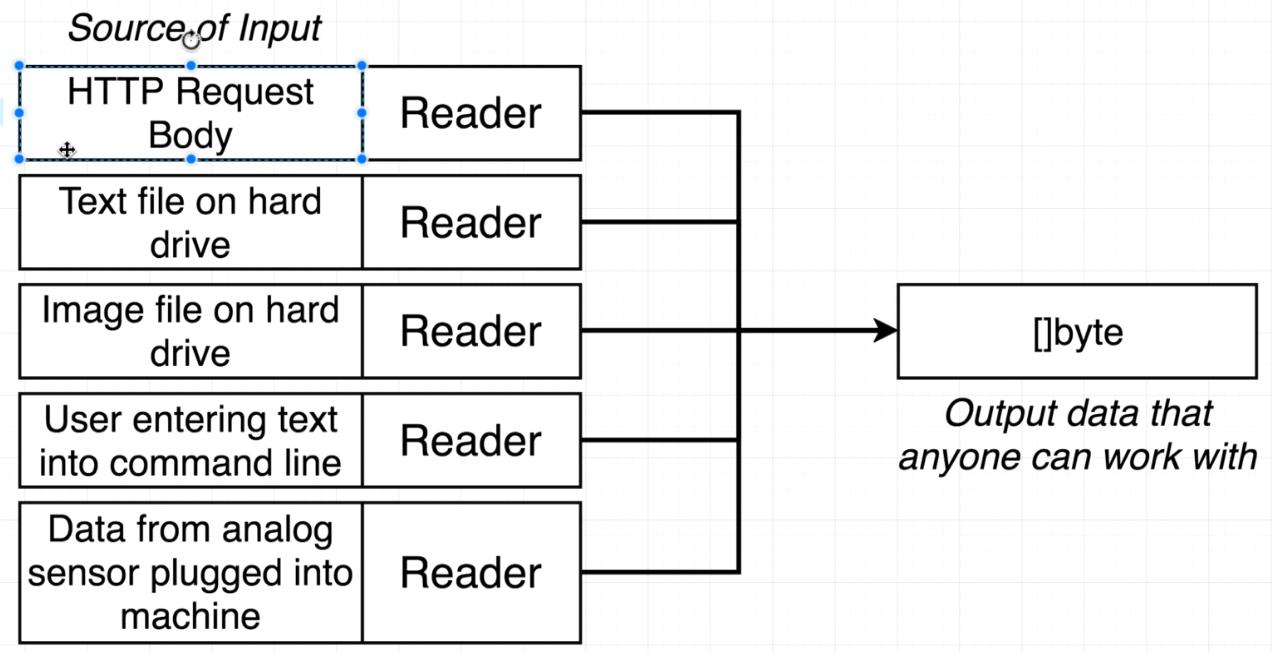
In Go, we can also assemble interfaces together to create new interfaces. Both Reader and Closer are interfaces. If you want to satisfy the requirements of the ReadCloser interface, then you have to satisfy the requirements of both the Reader interface and the Closer.

What really matters is that we satisfy what both the Reader interface and the Closer interface are requiring of us.

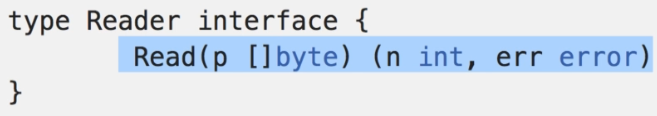


The purpose of the io.Reader interface is to give us some common point of contact for all these inputs that might come into our application. Whether your source is an HTTP request body, a text file, an image, command line text, or data from some analog sensor plugged into a machine, we can know that a function that satisfies the io.Reader interface will convert the source of input into a slice of bytes.

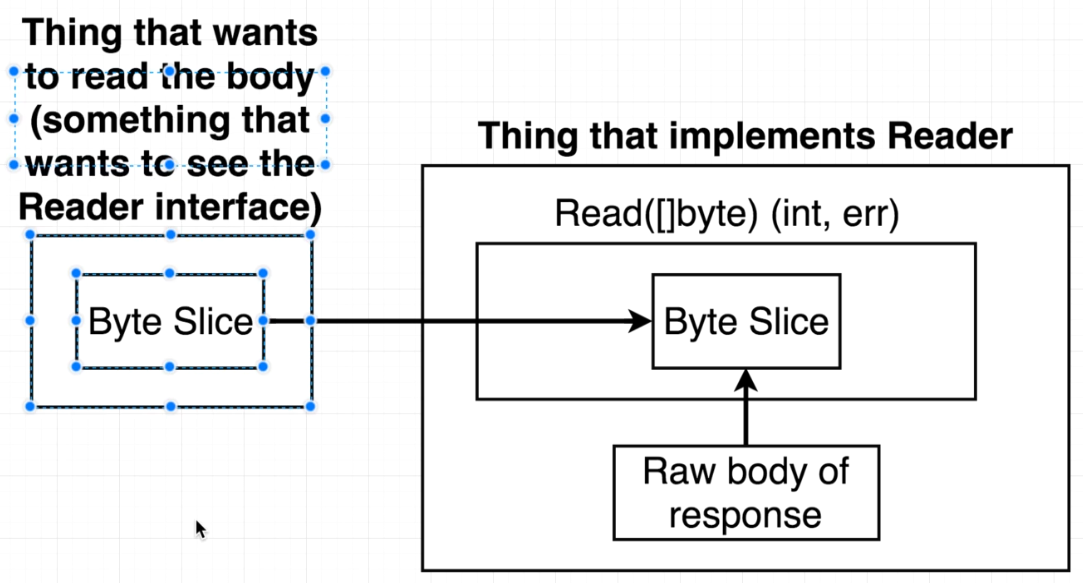
So, if every single source of data/input that comes into our application is going to be implementing the Reader interface.



Now, we just need to take advantage of the Reader interface and how to take the data out of the request body and do something useful with it.



This might be slightly confusing, because we want to receive a byte slice from the Reader, so why are we sending it a byte slice and getting back an int and an error? We are going to take advantage of pointers.



We are in the left of the above diagram. We are going to create our own byte slice, pass it to Read, the Read function will push some data into the byte slice, and then we will have data to make use of in our byte slice.

The int returned is the number of bytes read into the slice, and the error object will show if something went wrong.

This would be the primitive way to accomplish this:

// make http request to get the google home page

resp, err := http.Get("http://google.com")

// declare a byte slice with room for 99999 bytes

bs := make([]byte, 99999)

// call the Read function on the Body, which puts the http code into our byte slice

resp.Body.Read(bs)

fmt.Println(string(bs))

Here is a more ‘Go’-like method to accomplish the same thing:

// make http request to get the google home page

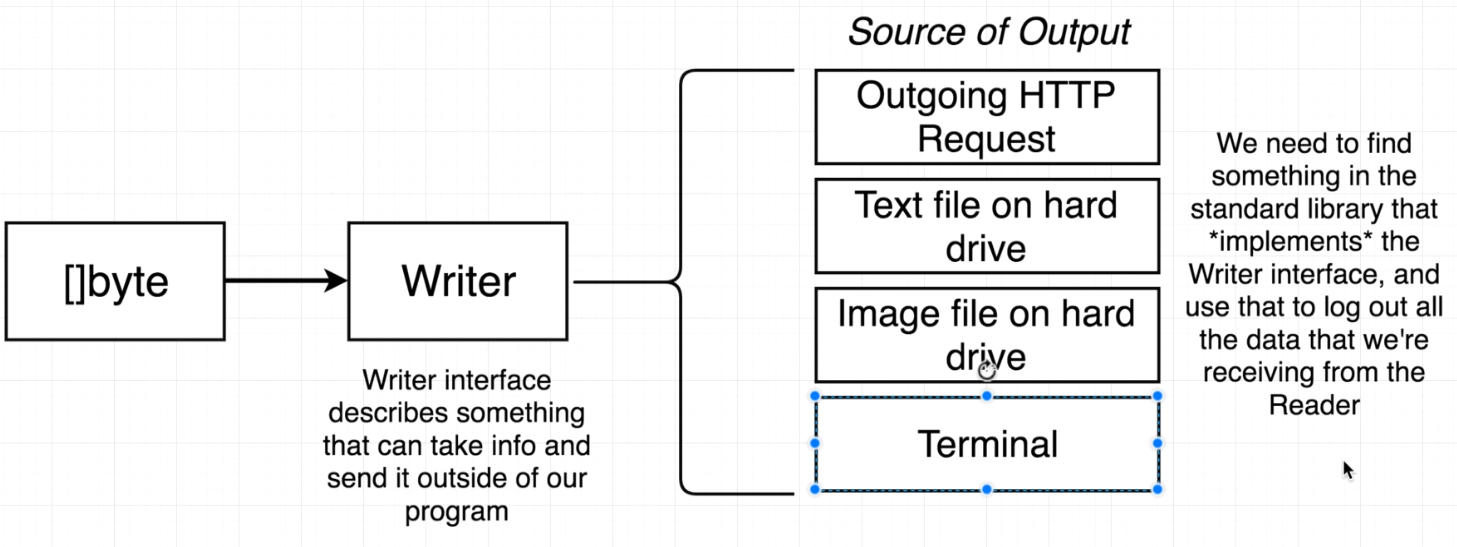
resp, err := http.Get("http://google.com")

// this accomplishes the same thing

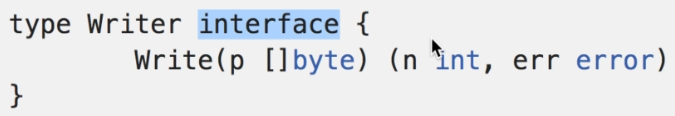
io.Copy(os.Stdout, resp.Body)

But what’s going on inside io.Copy()?

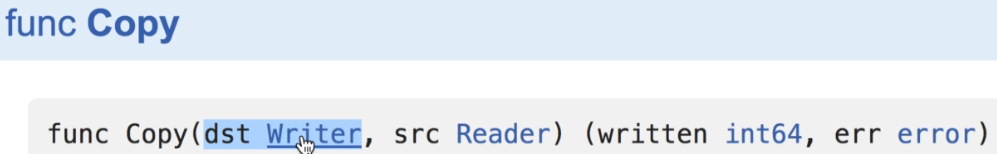
The purpose of the Writer interface is to take information and send it to some form of output that will work with some outside application.

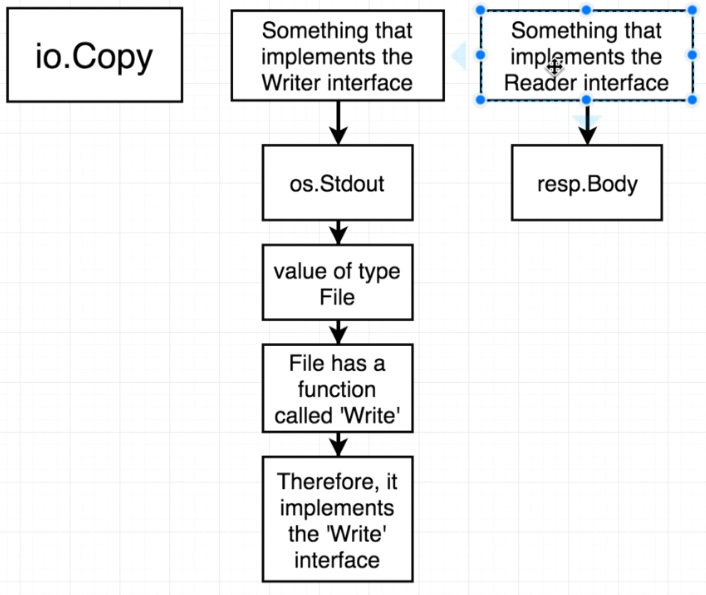


The signature for the Write() function looks a lot like the signature for the Read() function. In this case, the input is truly being used as an input. The data is taken in via the byte slice and then the information is sent to some output channel. Maybe that’s a text file, the terminal, or something else.

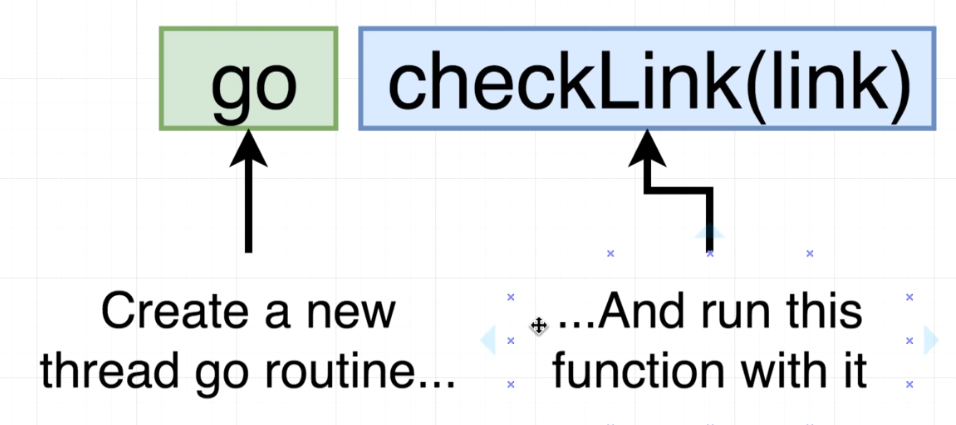


The Copy function takes some information from outside of our application (via the Reader) and then write/copy it to some outside channel (via the Writer).

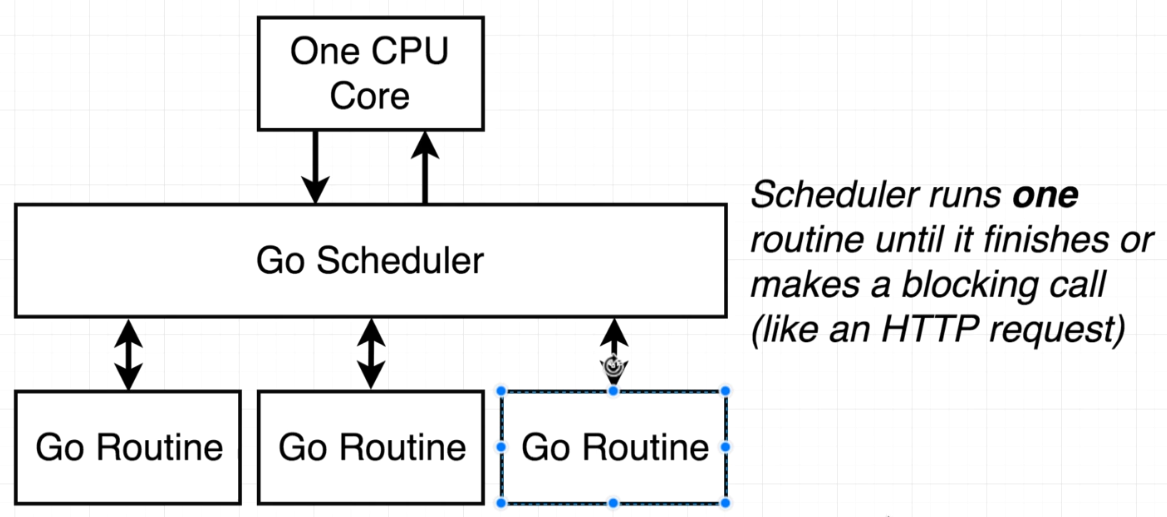




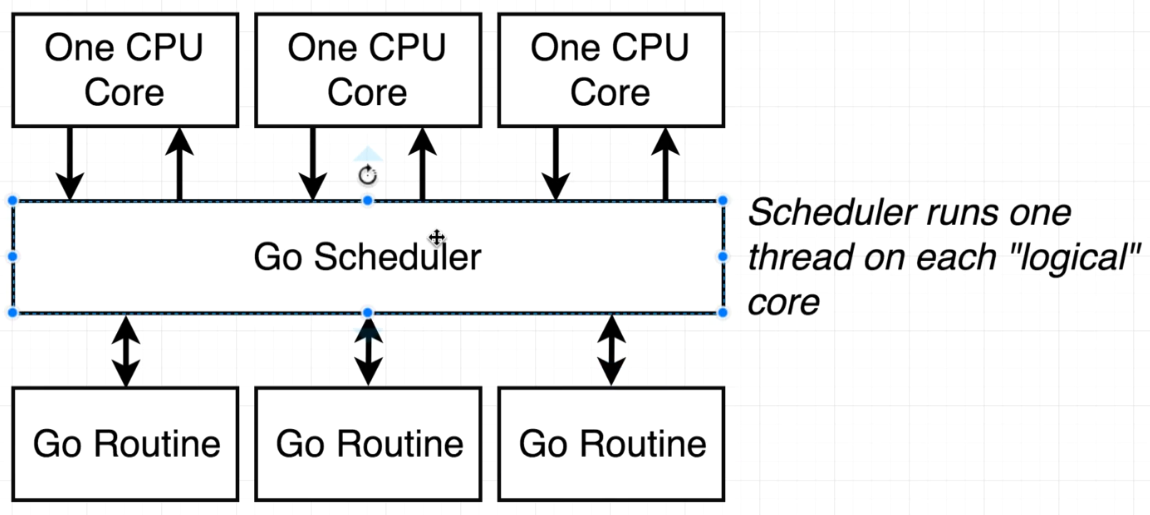
Go Routines & Channels



By default, the Go Scheduler tries to use one CPU. Only one go routine is running at a time, and the scheduler will pause a routine to run another one. Even though we’re spawning multiple routines, we’re only truly running one of them at a time if we’re on one CPU.

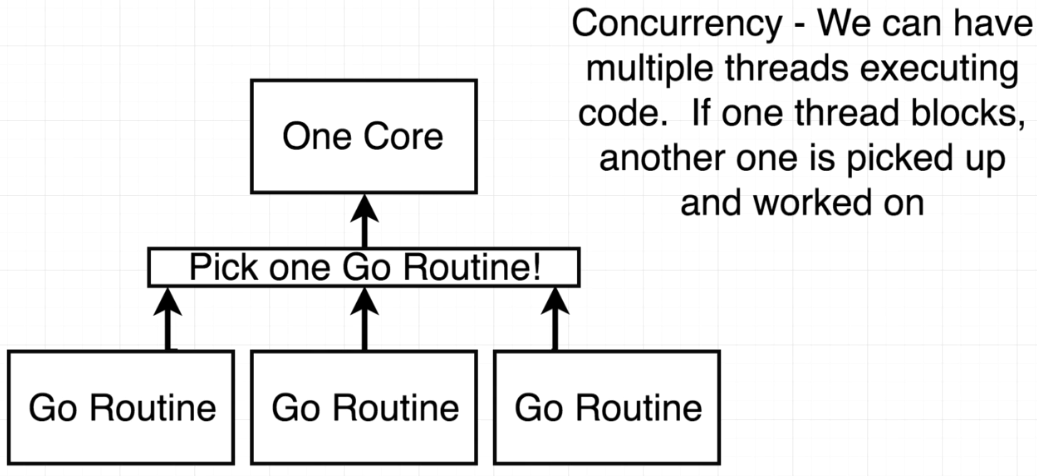


If we have more than one CPU Core on our machine and we override the setting, we can run a single go routine on each cpu core.

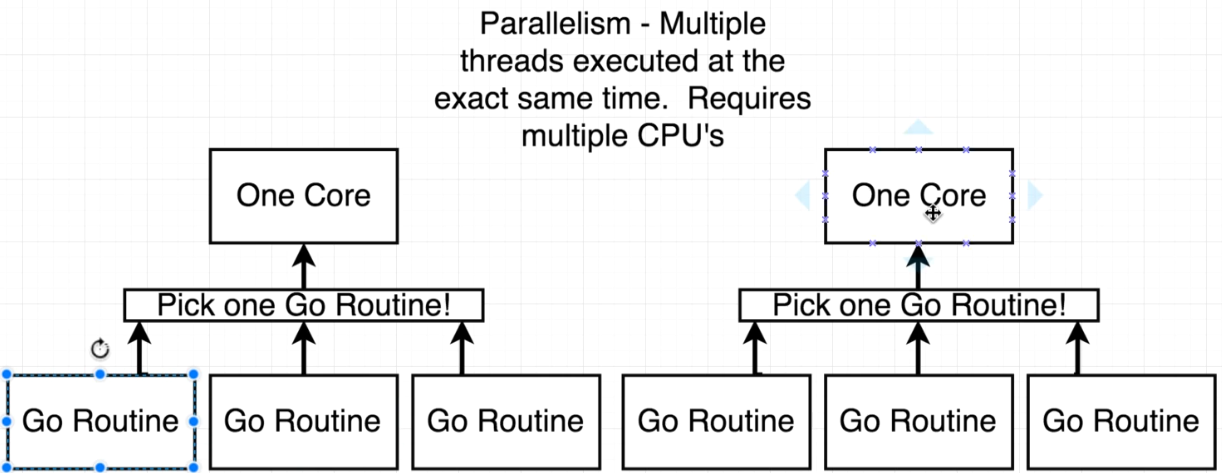


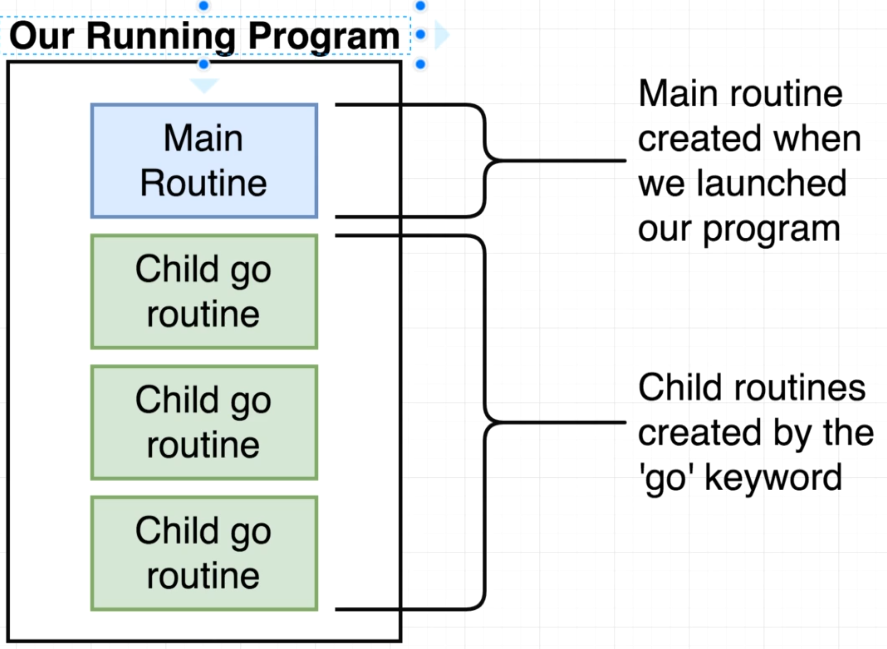
Remember that concurrency is not parallelism.

A program is **concurrent** if it can run multiple go routines (even if just on one core)



A program is only **parallel** if it can run multiple routines simultaneously (requires multiple cpu cores)

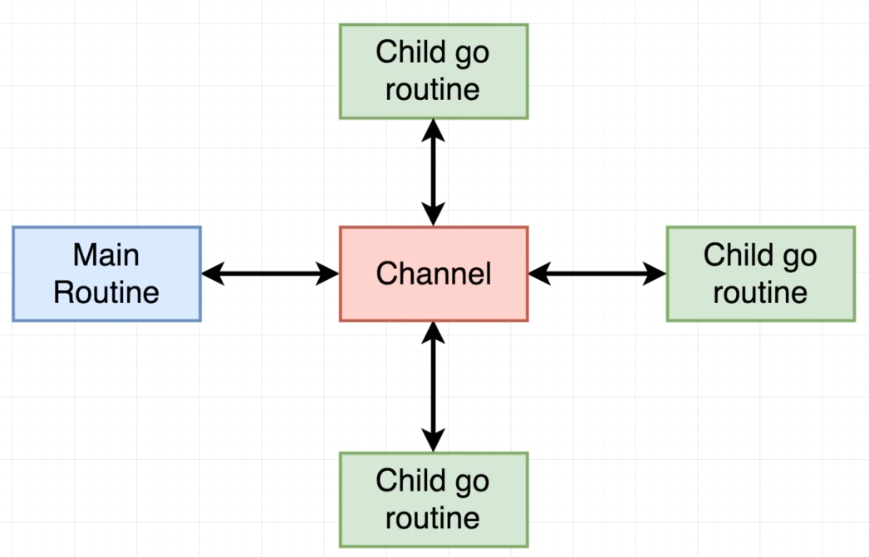




Child go routines don’t get the same level of “respect” as the main routine

We are going to use a channel to make sure that the main routine knows when each of the child routines have completed. We will create one channel, and it will communicate between these go routines

We can think of the channel as something like messaging between the routines



Channels are meant to share a specific type between routines

