# GO Workspaces:

In Go, programs are kept in a directory hierarchy that is called a workspace.

A workspace is root directory of your Go applications.

A workspace contains three subdirectories at its root:

src - source files organized as packages.

pkg - Go package objects.

bin - Compiled objects of each module executable programs, i.e., binaries.

When we do import "<package-name>" Go looks for these packages in $GOROOT/src or $GOPATH/src directories.

If a precompiled package object is found, Go uses that instead. Executables link all those compiled objects based on the modules used in the src code.

go install - creates package objects

go build – to test if the package compiles

go run - to compile and run

GOPATH - location of go workspaces.

Every go program should be part of a package.

If we define the package as package main, when we run such code, it creates an executable.

Every .go file in the same folder must have the same package name.

All identifiers will be exported to other packages if the first letter of the identifier name starts with an uppercase.

main function - entry point of execution

go.mod - first line is the name or URL or the import path of the go module

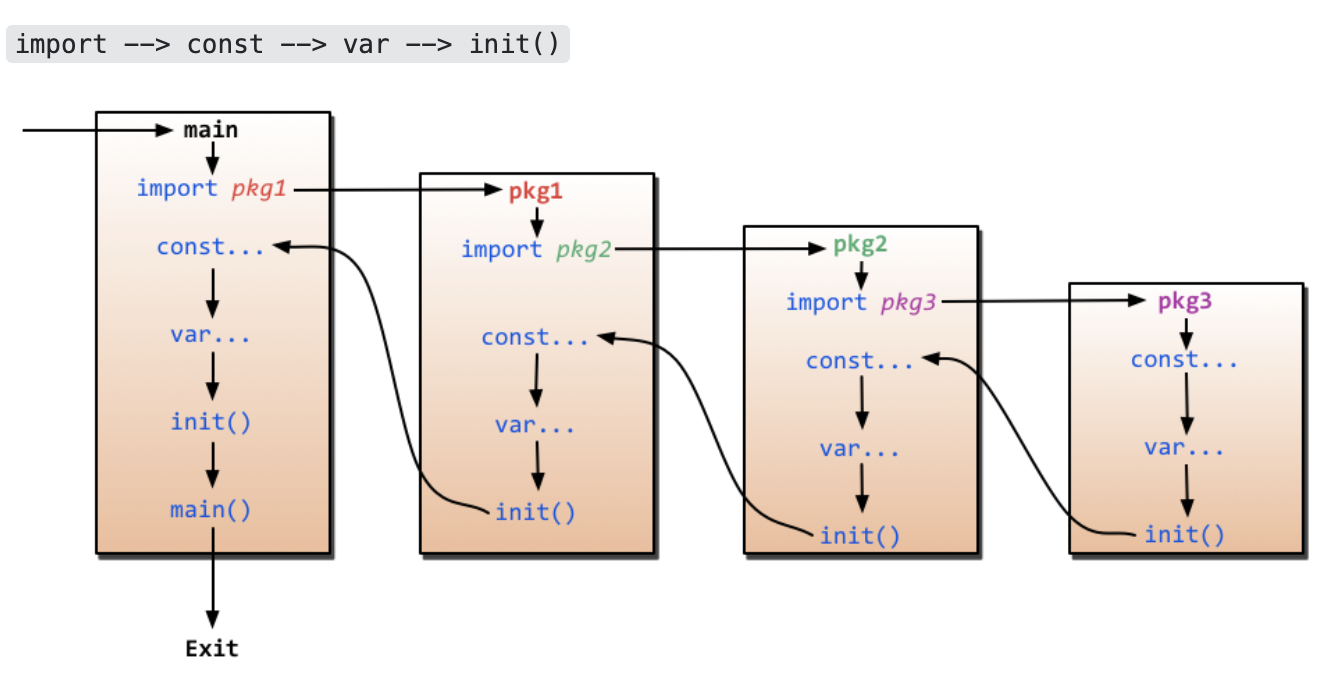
go mod tidy - adds any extra dependencies needed by the OS, architecture or the build.

go.sum is just for validation purposes. It has the expected cryptographic checksums of the content of specific module versions, so that we know if there is any change in the dependent module.

# init() function:

If we need to set some state on the initial start-up of the program, like creating connections to databases, or loading in configuration from locally stored configuration files etc, init() function is helpful. The sequence is  
a) variable declaration – const and then var  
b) init  
c) main

Even before all this, comes the imported packages order. Imported packages also follow the same order.



# Pointers:

Pointer - is a memory address to the variable it points to.

Go by default is pass by value, to change it to pass by reference, use pointers.

Pointers allows us to pass references to variables instead of passing the actual variables.

Passing the pointer allows us to effect the changes to the actual variable, whereas pass by value will not change the actual variable.

The default value of pointer is nil

Difference between pointer receiver and value receiver is:

value receiver - pass by value

pointer receiver - pass by reference

Meaning of pass by value:

When you pass an argument to a function, a copy of argument is created and that copy is passed to the function.   
 To pass the original value, use pointers. This helps in saving memory as we avoid creating unnecessary copies while calling functions.

We should compare value held by the pointer if we need a value comparison, pointer comparison would only tell if both variables point to the same value in memory or if they are nil. Though the value of two struct variables are the same, the pointer comparison might not be same.

Wildcard character \* is called the dereference operator in golang.

If you create a pointer to the struct fields can be accessed with that pointer.

# Functions:

A function with named return values can just do a simple “return” at the end, this would by default return the named return values in the function signature.

E.g.: func split(sum int) (x, y int) {  
 x = 0  
 y = 1  
 return  
 }

Variables declared in if block are available only inside if block.

The function arguments are evaluated before the function is called.

Functions are arguments too. They can be passed to other functions as arguments and also as return values.

**Switch Case:**switch item{  
case value1:  
case value2:  
default:  
}

switch {  
case bool1:  
case bool2:  
default:  
}  
  
The second switch syntax is like if, else if conditions.

# Methods:

func keyword can be used in two cases, one to create functions. Two, to create methods on types, here the types are called receivers.

In Golang, there are no classes. On the types, we can define methods. The type would be the receiver of the method.

Method = Function + receiver  
  
func (receiver receiverType) methodName(inpArguments) returnValueType

Only condition is that the receiverType must be in the same package in which the method is defined. This rule is true for int, float built-in types.

Pointer receivers are more common than value receivers because methods often have to modify the receivers.

# Interfaces:

1. Similar to Struct, where Struct holds variables, Interface holds method signatures.  
   Interface too is a type, it is an abstract type.  
   instead of variable names and types in struct, interface has the fields as method signature.  
   A method signature is name of   
   fnName(inp1 string, inp2 string) (string, string)  
   Any type(struct etc) which implements the methods of the interface are said to be implementing the interface, and are of the interface type. For the type to implement the methods of the interface, we need to define those types on the type as method and the type would be the receiver.
2. It is about behaviour of the object.
3. Interface is a tool for composition.   
   Composition: “has-a”, Inheritance: “is-a”
4. Tool for abstraction. Instead of designing our abstractions in terms of what kind of data our types can hold, we design our abstractions in terms of what actions our types can execute.
5. Interfaces make the function agnostic (or flexible) about the exact type of object it receives. All that matters is what methods it has. In other words, if an object satisfies an interface, we can be sure that it implements all the methods defined in the interface.
6. a pointer type may call the methods of its associated value type, but not vice versa
7. Interfaces specify behaviour (read methods), not data
8. Use of interfaces – if we pass interface to a function argument, it doesn’t matter what type of underlying variable, what matters is if the argument satisfies the interface’s methods or not. So inside the function we can safely call the interface’s methods on the arguments, without being bothered if the method is implemented by the argument or the type of the argument. interface{}

Interface helps to achieve run time polymorphism, i.e., call is resolved at run time.

An interface is a set of methods, and also a type.  
Instead of designing abstractions in terms of what kind of data our types can hold, we design our abstractions in terms of what actions our types can execute.  
Interface has two things:

a pointer to information about the type stored

a pointer to the associated data

An interface value is nil only if both the above values are nil.

int, string, struct are all concrete data types, we know the data type and know what operations can be done on that data type, but interface is **abstract** type , we get to know the type and value dynamically. The struct which implements this interface, it’s type and value is the interface’s type and value respectively.

Empty interface:

interface with zero methods. It is a general container type because all types satisfy the empty interface.

interface holds reference to the underlying struct, similar to how a slice holds the reference to an array.   
Syntax:

var i SampleInterface  
i = SampleStructWhichImplementsInterface(StructValues)  
fmt.Printf("(%v, %T)\n", i, i)  
would give (StructValues, SampleStructWhichImplementsInterface)

var v interface{} 🡪 here ‘v’ is not any type, it is interface{} type. V is empty interface, both type and value would be nil, till we assign v = SampleStruct(StructValues)

The type of interface is always the struct which implements that interface.  
When a function is defined as func xyx(v interface{}) ,   
Go runtime converts the passed argument value to the type interface{}.

Most of the time, we do not need to know the type of the interface, all we need is to just call the methods defined in the interface, that’s it.

1. There are no exceptions in go, only errors
2. Defers happen Last in First out
3. Recover is only helpful with Defers. Defer + anonymous fn + recover
4. Though Go is statically typed language, while declaring a variable you can omit the data type of it, as go uses type inference. It infers the datatype from the data we are passing to variable.
5. In Go, variables are always initialized to a well-defined value, if no value is specified, the zero value is picked.
6. Difference between array and slice  
    Array is immutable.

Slice syntax - var mySlice3 := make([]int, 4, 4)

1. Polymorphism in Go is achieved with the help of interfaces.

interface can act as two things:

Collection of method signatures

Custom types  
The interfaces are abstract which is why we cannot create its instance

# Struct:

Struct is a datatype in golang which contains the blueprint of data the structure will hold.

Used to store Records.

The data consists of field names and the datatypes of the fields

###############################################################################

Struct is called a literal. Literal is a syntactic way to construct a data structure. We need to know beforehand, the possible field names in the literal, we cannot construct it by creating an empty data structure and adding or subtracting fields as we go.

The struct fields can be accessed by the struct pointer too with the . notation.

Eg:   
p = &StructSample  
p.StructField is same as StructSample.StructField

###############################################################################

Variadic functions

A variadic function accepts an infinite number of arguments and all these arguments are stored in a slice.

###############################################################################

Slices and Arrays differences

Slices are sequences of typed data. They are abstractions on top of array. Hence they do not store any data as they are abstractions.

Array elements are indexed. Arrays are inflexible as they have fixed size. So an Abstraction is built on top of array which is Slice.

A slice is not an array. A slice describes a piece of an array.

Slice points to the underlying array and the capacity of slice is the size of the array.

Array definition - [3]bool{true, true, false}  
Slice definition - []bool{true, true, false}

Slice has 3 elements:  
1. Pointer to the underlying array  
2. Length of the slice  
3. Capacity of the slice.  
  
Length – no.of elements of slice  
Capacity – no.of elements of underlying array

So slicing operation in slice is as good as accessing array elements.

If you want to go beyond the slice capacity, you need to use copy function.

Fields and methods in a struct – has-a relation

Embedded types – is-a relation

Type embedding:  
The main purpose of type embedding is to extend the functionalities of the embedded types into the embedding type, so that we don't need to re-implement the functionalities of the embedded types for the embedding type. The embedded field’s methods can be invoked directly on the struct variable.

# Inheritance, Composition and Aggregation:

is-a: Inheritance – Derived – tightly coupled

Association:  
has-a: Aggregation – both the lifecycles are independent  
owns-a: Composition – lifecycles are coupled and contained entity cannot live without other.

Use association over inheritance.

# Slice:

Slice is same as array, whereas array has a predefined length, slice doesn’t have length defined in it’s declaration.

Slice has both length and capacity:  
Length = number of elements it contains.

Capacity = number of elements in the underlying array

Syntax:

s := []int{2, 3, 5, 7, 11, 13}  
var s []int

s[0] = 0

s := make([]int, 0, 5) // with make, len and cap

s = append(s, 1, 2, 3, 4)

# Maps

Maps are associative data types.

The biggest problem in golang is to differentiate between actual zero value and the default zero value.

# Methods:

Methods and Functions differences

var\_name.method\_name()

function\_name(var\_name)

Same method\_name can be used for different variables, but the function name should be unique.

Methods can be chained.

a pointer type may call the methods of its associated value type, but not vice versa

# Naming Convention:

Package names – lowercase single word  
Files: all lower\_case, separated with “\_”

Starting with “.” Or “\_” are ignored

Ending with \_test are run by go test

Functions: CamelCase

Constants: All CAPS, separated with “\_”

Variables: Boolean variables should start with Has, Is etc.

Identifier, is a fancy word for variable name.

Short variable names work well when the distance between their declaration and last use is short.

Use long variable names, if the variable is used in multiple places.

Don’t include the name of the type in the name of your variable.

Constants should describe the value they hold, not how that value is used.

Single letter variables for loops and branches  
Single words for parameters and return values  
Multiple words for functions and package level declarations  
Single words for methods, interfaces, and packages.

**Comments:**

Comments should explain What, How and Why.

Comment on a variable - comment should describe the variables contents, not the variables purpose

Always return early rather than nesting deep.

# OmitEmpty:

<https://www.sohamkamani.com/golang/omitempty/>

used for json encoding.

Needed when we want to differentiate between default values and actual values. Example – default value of int is 0, if we do not use omitEmpty, we cannot know if the actual value of the struct field is zero or is it set to the default value. If we use omitempty, the default value fields would be omitted while json encoded object.

# Lock:

var m sync.Mutex  
m.Lock()  
# Your piece of code  
m.Unlock()  
  
**Mutexes vs Channels:**

Mutex – sequential access of a resource

Channels – orchestrating

Mutexes have performance advantage over channels.  
Channels are best suited in cases like passing ownership of data, distributing units of work, and communicating async results.  
channels have in built thread safety and prevent race conditions. Using mutexes is helpful when you just need locks over a few shared resources.

# Channels:

Channels are communication channels for goroutines.

You can write to the channels and wait on the channel to read input from the channel.

If you read from empty channel, it's a deadlock.

Channel read and write are blocking operations.  
When we send data into the channel using a GoRoutine, it will be blocked until the data is consumed by another GoRoutine. When we receive data from channel using a GoRoutine, it will be blocked until the data is available in the channel.

Follow the direction of the arrow to understand Reads and Writes.

msg := <-messages // Channel Read

messages <- "ping" // Channel Write

## Buffered channels:

Difference between channels and Buffered Channels:

Buffer size: the number of elements that can be sent to the channel without the send blocking. By default, a channel has a buffer size of 0 (you get this with make(chan int)). This means that every single send will block until another goroutine receives from the channel. A channel of buffer size 1 can hold 1 element until sending blocks.

Channels are blocking as they only store one item, Only one item can be read or write at a time.  
If you want to send a value to the channel, it is blocked till another goroutine receives this value. For buffered channel you have a buffer of values. Only if the buffer is full, it gets blocked, i.e., deadlock.

A buffered channel with full buffer elements behaves the same as a normal channel.

Select statements lets you wait on channel operations.  
To have non-blocking channel operations, use select with default, default case would be non-blocking.

It is a deadlock when all goroutines are blocked, If at least one of the main thread is functional, it would not cause deadlock

Difference between channel and buffered channel is that, the channel read and write operations are blocking in nature. Whereas buffered channel operations will not be blocked until it reaches the buffer length. Example – If you read from an empty channel or write to a full channel.

## WaitGroups: Useful if we need to block pieces of code. WaitGroup - Struct type Three functions: Add(int), Wait() and Done()

WaitGroup has an internal counter. Based on the integer value you pass to Add, the counter increases by that value. Once Done() is called the counter decreases by one. Wait() would wait for all the counters to be Done(). There should be enough Done() for each Add(). Done()s can be more but not Add()s.

WatiGroups help to wait till we finish all tasks.

## Deadlock and RaceCondition:

Race Condition:

Two transactions T1 and T2. They reach out for the same shared variable. Both are in different threads. The last transaction which changes the value of the shared variable, that is the final value of that variable. Eg- T1 and T2 read the variable value as x. T1 changes the value to y, and then T2 acquires the processor time and changes the value to z. So the final value of variable is now z.

Solution: Lock the critical section as an atomic instruction.

Deadlock:  
 A deadlock occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked. Thread T1 locks variable x, thread T2 locks variable y. Now T1 needs y and T2 needs x. T1 waits on T2 for the lock on y to release and T2 for x. This is a case of cyclic dependency. When a thread waits on another thread which waits on another thread, then deadlock occurs. As a result, each thread stops executing and waits for the other thread to release the variable. Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked.  
Solution: Lock the variables in the same order

Routing in golang

To implement a server, we need two things: Port and routes.

flow of control for a HTTP request looks like:

Router => Middleware Handler => Application Handler

1. Router in golang - ServeMux

which is a HTTP request router, It matches the URL of each incoming request against a list of

registered patterns and calls the handler for the pattern that most closely matches the URL.

2. Middleware - Shared functionality that you want to run for many (or even all) HTTP requests.

Middleware should satisfy http.Handler interface

Build a chain of handlers containing both middleware handler and normal application handler,

which can be registered with a router.

3. Handlers are responsible for writing response headers and bodies.

Any object can be a handler, so long as it satisfies the http.Handler interface.

It must have a ServeHTTP method with the following signature: ServeHTTP(http.ResponseWriter, \*http.Request)

###############################################################################

Errors:

An error is return value in golang which is returned if something unexpected happened.

"error" is builtin type in golang.

type error interface {

Error() string

}

How to create a new error type:

any type that implements Error() function is an "error" type.

errors.New("Custom error type")

###############################################################################

Marshalling, Unmarshalling <---> Encoding, Decoding:

Marshalling - Go objects to JSON, It returns a []byte of JSON data  
Unmarshalling – reverse of marshal.

Encoding - Encoding, first marshals the object to a JSON encoded string, then writes that data to a buffer stream. Encode helps in sending data to server, decoder helps in receiving data from server.

###############################################################################

Goroutines

Goroutines are cheap. Because of the runtime’s ability to multiplex goroutines onto a small pool of threads (which you don’t have to manage), hundreds of thousands, millions of goroutines are easily accommodated.

###############################################################################

Panic Defer Recover

Defer gets executed just before the function returns or in case of panic.  
Defer statement defers the execution of a function until the surrounding function returns.

###############################################################################

**CONCURRENCY:**

Concurrency is handling multiple things at same time.

Concurrency is DEALING with multiple things at once,   
Parallelism is DOING multiple things at once.

Concurrent programming is - programs fight for the cpu time

Parallel programming - independent CPU times are available for the programs.

Parallel programming is to specifically refer to the simultaneous execution of concurrent tasks on different processors or cores.

Thus, all parallel programming is concurrent, but not all concurrent programming is parallel.

Reference - https://www.mineiros.io/blog/guide-to-multihreading-and-multiprocessing

What is a Process:

A process is an instance of a program.

A computer program in languages like C, java or Go, it is just a text file. As the computer only understands binary instructions composed of 0s and 1s, the code needs to be compiled to machine language. When a compiled program is sent to OS to handle, OS allocates:

memory address space (where process’s heap and stacks will be located),

program counter,

PID (process id).

A process has at least one thread known as primary thread, primary thread can create multiple other threads. When the primary thread is done with its execution, process exits.

In nutshell, a process is a program in the memory.

Thread:

Thread is a light-weight process inside a process. It is the actual executor of a piece of code. It has access to memory provided by the process, OS resources, and other things.

A stack is created at runtime and is normally of a fixed size, preferably 1-2 MB. While the stack of a thread can be used by only that thread and will not be shared with other thread. A heap is a property of a process and it is available to use by any thread. Heap is a shared memory space where data from one thread can be access by other threads as well. Threads share the heap, but can have their own stack.

# OOPs in golang:

Encapsulation – Package Level,  
Go encapsulates things at the package level. Names that start with a lowercase letter are only visible within that package. You can hide anything in a private package and just expose specific types, interfaces, and factory functions.

Inheritance – Type Embedding,   
composition by embedding an anonymous type is equivalent to implementation inheritance.

Polymorphism – Interface,   
A variable of type interface can hold any value which implements the interface. This property of interfaces is used to achieve polymorphism in Go. Compile time polymorphism is not possible in golang as the function names cannot be defined twice.

# Testing in golang:

go test -v  
go test -coverprofile=coverage.out  
go tool cover -html=coverage.out

func TestAdd(t \*testing.T){

got := Add(4, 6)

want := 10

if got != want {

t.Errorf("got %q, wanted %q", got, want)

}

}

// arg1 means argument 1 and arg2 means argument 2, and the expected stands for the 'result we expect'

type addTest struct {

arg1, arg2, expected int

}

var addTests = []addTest{

addTest{2, 3, 5},

addTest{4, 8, 12},

addTest{6, 9, 15},

addTest{3, 10, 13},

}

func TestAdd(t \*testing.T){

for \_, test := range addTests{

if output := Add(test.arg1, test.arg2); output != test.expected {

t.Errorf("Output %q not equal to expected %q", output, test.expected)

}

}

}

# References:

1. <https://jordanorelli.com/post/32665860244/how-to-use-interfaces-in-go>
2. <https://go.dev/play/p/Q0xdQc-CJJK>
3. <https://go101.org/article/type-embedding.html> - struct field embedding
4. <https://octo.vmware.com/golang-embedding/>
5. <https://go.dev/doc/faq#Is_Go_an_object-oriented_language>