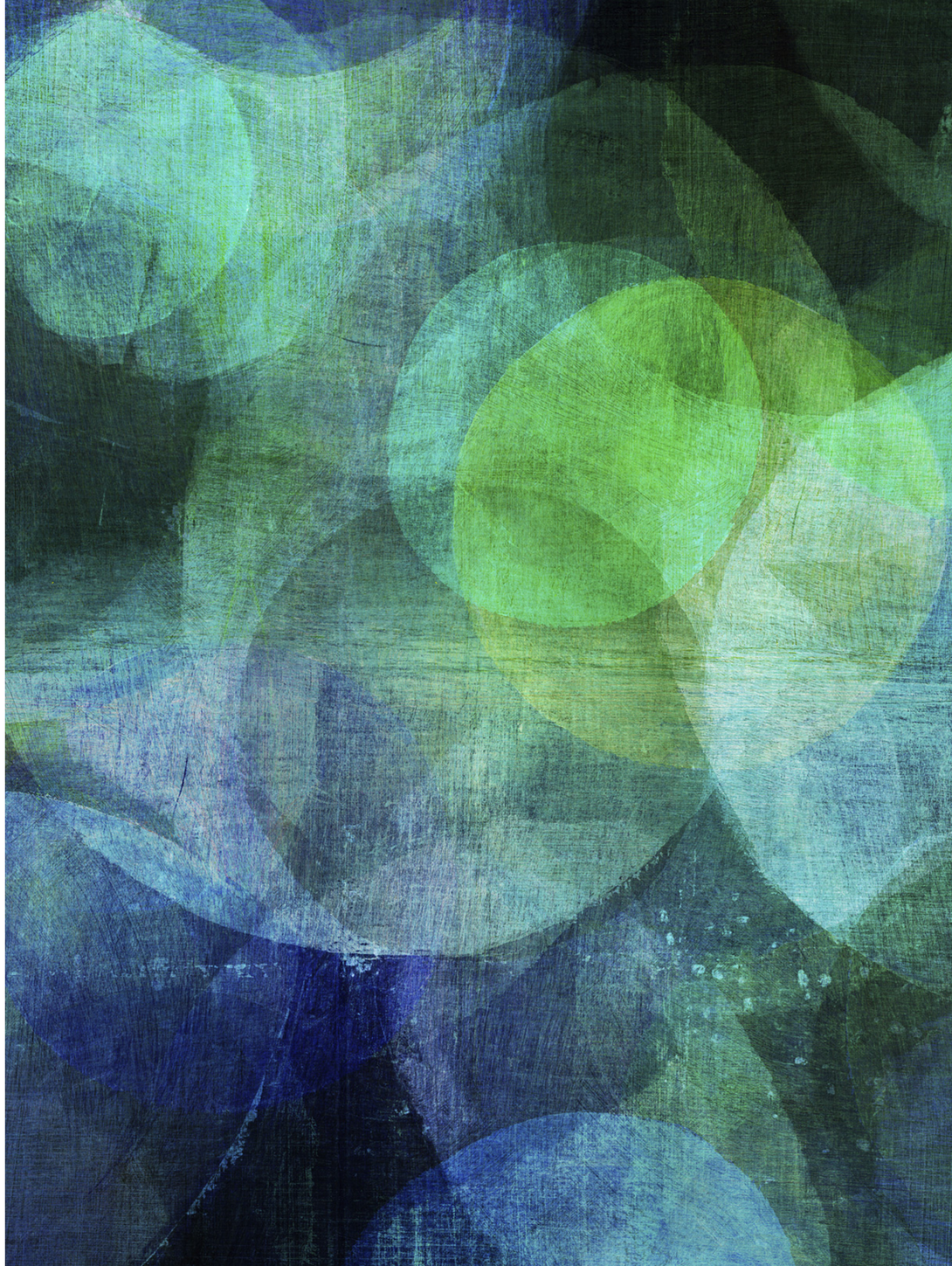
GOLANG TRAINING

by Jaiprakash Singh

Introduction:

1. Intro
2. Features
3. Setup
4. Hello World
5. Packages
6. Modules
7. Go Env



INTRODUCTION

- The Go programming language is an open source project to make programmers more productive.
- Go was developed at Google.
- Go is expressive, concise, clean, and efficient.
- Go concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction.
- Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection.
- Go is a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.

GO TOOLS

- build - compiles packages and dependencies
- run - compiles and run Go program
- clean - remove object files
- env - print Go environment information
- test - test packages and benchmarks
- Others (fmt, fix, get, install, list, tool, version, vet)

GO FEATURES

- Compiled
- Garbage-collected
- Has own runtime
- Simple syntax
- Great standard library
- Cross-platform
- Object Oriented (no inheritance)
- Statically and strongly types
- Concurrent (goroutines)
- Closures
- Explicit Dependencies
- Multiple return values
- Pointers and more

FEATURES NOT IN GO

- Exception handling
- Inheritance
- Generics
- Assert
- Method overloading

GO PACKAGES AND MODULES

- Go Package: A package is nothing but a directory inside your Go workspace containing one or more Go source files, or other Go packages. Every Go source file belongs to a package.
- Go Module: A *module* is a collection of packages that are released, versioned, and distributed together. Modules may be downloaded directly from version control repositories or from module proxy servers. A module is identified by a module path, which is declared in a `go.mod` file, together with information about the module's dependencies.

GO ENV

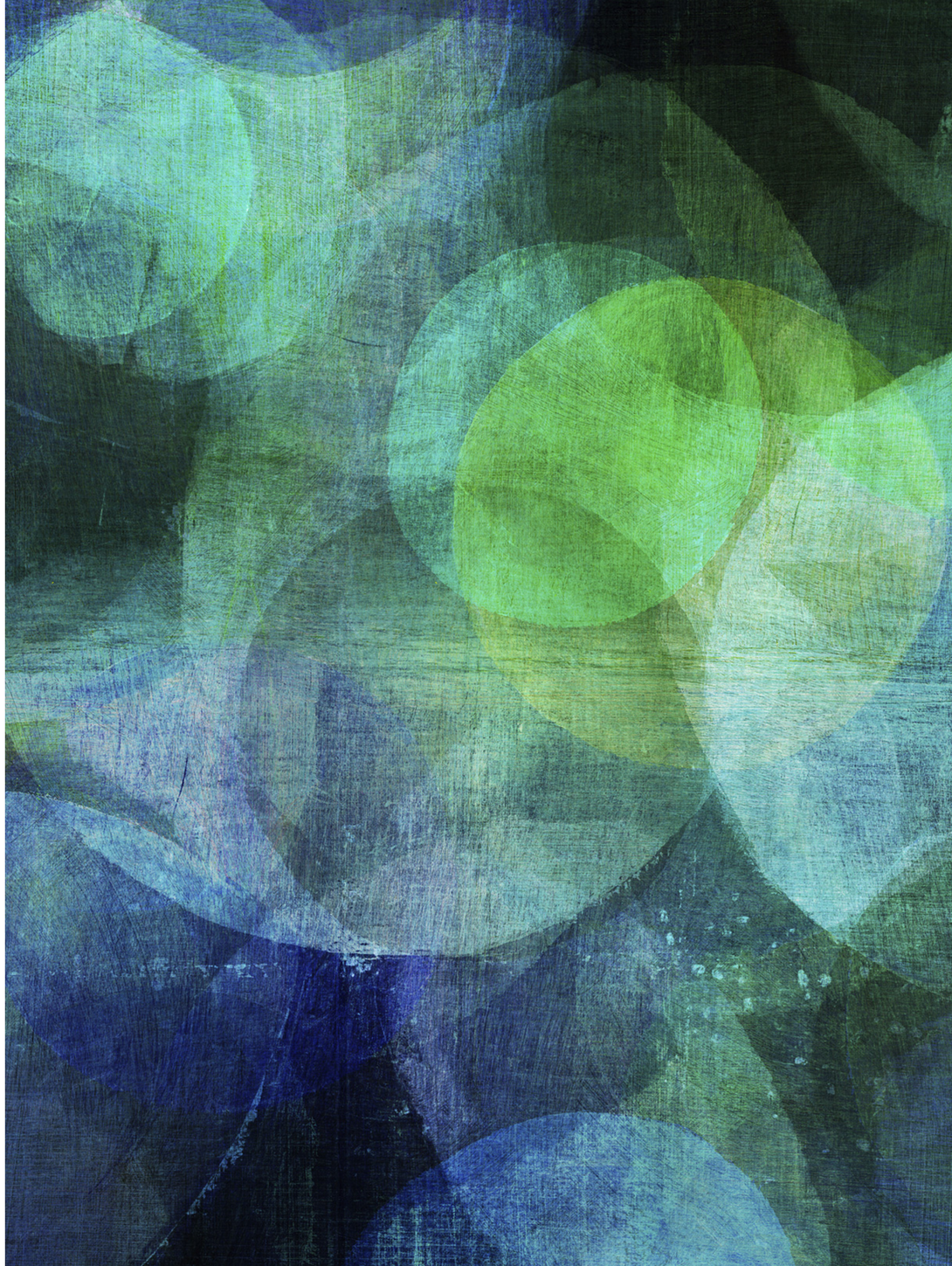
- **GOPATH**: The GOPATH is the home path where the Go code resides. The GOPATH is used to resolve imports, as well as to install packages outside the go tree.
- GOPATH contains 3 directories under it and each directory under it has specific functions:
  - **src:** It holds source code. The path below this directory determines the import path or the executable name.
  - **pkg:** It holds installed package objects. Each target operating system and architecture pair has its own subdirectory of pkg.
  - **bin:** It holds compiled commands. Every command is named for its source directory.
- **GOROOT**: GOROOT is the place where the go installation took place. GOROOT is for compiler and tools that come from go installation and is used to find the standard libraries. It should always be set to the installation directory.
- **GOBIN**: GOBIN is the directory where `go install` and `go get` will place binaries after building `main` packages.
- **GOOS**: GOOS is used for cross platform compilation. Possible values are linux, darwin, windows, android, etc.
- **GOARCH**: GOARCH is used for cross platform compilation. Possible values are 386, amd64, arm64, etc.

PRACTICE & REFERENCE (BASIC GOLANG):

- https://tour.golang.org/welcome/1
- https://golangbyexample.com/packages-modules-go-first/
- https://golangbyexample.com/packages-modules-go-second/
- https://gobyexample.com/

Basics:

1. In-built Data types
2. Custom Types
3. Variable Scope
4. Arrays
5. Slices
6. Maps
7. Structs
8. Functions
9. Control statements



IN-BUILT DATA TYPES

- **Primitive Data Types:**
  - Numbers: int, int8, int16, int32, int64, float32, float64, complex64, complex128
  - String
  - Boolean
  - Byte
  - Rune
- **Composite Types:**
  - **Non-Reference Types**
    - Collection
      - Arrays
    - Aggregation
      - Structs
  - **Reference Types**
    - Slices
    - Maps
    - Function/Methods
    - Channels
    - Pointers
  - **Special Types**
    - Interface

CUSTOM TYPES

- Type Declaration:

  var <variable> <data type> [= <initial value>]

  or

  <variable> := <initial value>

  ```
  var x int = 2
  var y int
  y = 1
  z := 5
  var name string = "Tom"
  var isValid bool = false
  var a byte = 97
  b := 'b'
  ```

  - Custom type:

    ```
    type engSize uint8
    var size uint8 = 6
    var fordEng engSize = size
    ```

CUSTOM TYPES

- Type Declaration:
  - Custom type:

```
type engSize uint8
var size uint8 = 6
var fordEng engSize = size
```

  - Struct type:

```
type vehicle struct {
   make, model string
   engine engSize
}
```

- Type Conversion:
  - Basic type:

```
var count int32
var actual int
var test int64 = actual + count  //invalid operation: actual + count (mismatched types int and int32)
var test int64 = int64(actual) + int64(count) //conversion expression
```

  - Custom type:

```
type engSize uint8
var size uint8 = 6
var fordEng engSize = engSize(size) //conversion expression
```

VARIABLE SCOPE

- Local and Global Variable
  - **Local Variable:**
    - Local variables are variables which are defined within a block or a function level
    - These variables are only be accessed from within their block or function
    - These variables only live till the end of the block or a function in which they are declared. After that, they are Garbage Collected.
  - **Global Variable:**
    - A variable will be global within a package if it is declared at the top of a file outside the scope of any function or block.
    - Global variable are available throughout the lifetime of a program.
    - **Global Variable Types:**
      - Exported Variables: If the variable name stats with an uppercase letter then it can be accessed from outside different package other than which it is declared.
      - Non-Exported Variables: If this variable name starts with a lowercase letter then it can be accessed from within the the package which contains this variable definition.

ARRAYS

- An array is a data structure that consists of a collection of elements of a single type which can hold more than one value at a time.
- An array holds a specific number of elements, and it cannot grow or shrink(cannot be resized).
- Go’s arrays are values. An array variable denotes the entire array; it is not a pointer to the first array element (as would be the case in C). This means that when you assign or pass around an array value you will make a copy of its contents.
- Arrays are a bit inflexible, so you don't see them too often in Go code.
- Array Operations:

```
b := [2]string{}
b[0] = "Penn"
b[1] = "Teller"
```

or

```
b := [2]string{"Penn", "Teller"}
```

or

```
b := [...]string{"Penn", “Teller"}
```

SLICES

- A slice is a flexible and extensible data structure to implement and manage collections of data. Slices are made up of multiple elements, all of the same type.
- Slice type provides a convenient and efficient means of working with sequences of typed data.
- A slice is a segment of dynamic arrays that can grow and shrink as you see fit. Like arrays, slices are index-able and have a length. Slices have a capacity and length property.
- A slice does not store any data, it just describes a section of an underlying array. Changing the elements of a slice modifies the corresponding elements of its underlying array. Other slices that share the same underlying array will see those changes.
- In practice, slices are much more common than arrays.
- Slice Operations:

```
letters := []string{"a", "b", "c", "d"}
```

or

```
s := make([]int, 5)
s = append(s, 1, 2, 3)
```

or

```
var arr = [4]int{1, 2, 3, 4}
var arrSlice = arr[1:3]
```

MAPS

- A map is a data structure that provides you with an unordered collection of key/value pairs.
- Maps are used to look up a value by its associated key. You store values into the map based on a key.
- The strength of a map is its ability to retrieve data quickly based on the key. A key works like an index, pointing to the value you associate with that key. A map is implemented using a hash table, which is providing faster lookups on the data element and you can easily retrieve a value by providing the key.
- Maps are unordered collections, and there's no way to predict the order in which the key/value pairs will be returned. Every iteration over a map could return a different order.
- Maps are passed as reference to functions.
- Map Operations:

```
var employee = map[string]int{"Mark": 10, "Sandy": 20}
```

or

```
var employee = make(map[string]int)
employee["Mark"] = 10
fmt.Println(employee["Mark"])
delete(employee, "Mark")
```

STRUCTS

- A **struct** is a **user-defined** type that represents a **collection** of **fields**. It can be used in places where it makes sense to group the data into a single unit rather than having each of them as separate values.
- A struct consists of both **built-in** and **user-defined types.**
- You can think of **struct** as a **class** but without **inheritance**.
- **Struct** can have methods defined just like a class.
- **Structs** in golang can contain other user-defined types as well. So, a struct can contain other **nested** structs.

STRUCTS EXAMPLE

```
package main
 import (
   "fmt"
)
 type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}
 func main() {
    //creating struct specifying field names
   emp1 := Employee{
      firstName: "Sam",
      age:       25,
      salary:    500,
      lastName:  “Anderson",
      }
    //creating struct without specifying field names
    emp2 := Employee{"Thomas", "Paul", 29, 800}
    fmt.Println("Employee 1", emp1)
    fmt.Println("Employee 2", emp2)
    fmt.Println("First Name:", emp1.firstName)
    fmt.Println("Last Name:", emp1.lastName)
    fmt.Println("Age:", emp1.age)
    fmt.Printf("Salary: $%d\n", emp1.salary)
}
```

FUNCTIONS & RETURN TYPES

- A function is a group of statements that together perform a task. Every Go program has at least one function, which is **main()**. You can divide your code into separate functions.
- A Go function can return multiple values.
- Function Return Type:
  - Single Return value:

```
func Swap(x int, y int) int {}
val := Swap(1,2)
```

  - Multiple Return value:

```
func swap(x int, y int) (int, int) {}
a, b := Swap(1,2)
```

- Function Call Type:
  - Call by value:

```
func swap(x int, y int) int {}
```

  - Call by reference:

```
func swap(x *int, y *int) {}
```

FUNCTIONS & RETURN TYPES

- Function Usage:
  - Function as Function:

```
func Swap(x int, y int) int { … }
Swap(1, 3)
```

  - Function as Value:

```
Swap := func(x int, y int) int { … }
swap()
```

  - Function Closure:

```
func getSwap() func(x int, y int) int { … }
Swap := getSwap()
Swap()
```

  - Variadic Function:

```
func Swap(x …int) int { … }
Swap(1, 2, 3)
```

  - Method:

```
type Swapper struct { … }
func (s Swapper) Swap(x int, y int) int { … }
Swapper.Swap()
```

CONTROL STATEMENTS (IF ELSE)

- If Statement

```
if condition { ... }
```

- If Else Statement

```
if condition {
    ...
} else {
    ...
}
```

- If Else If Statement

```
if condition1 {
    ...
} else if condition2 {
    ...
} else {
    ...
}
```

- If with short Statement

```
if a := 6; a > 5 { ... }
```

CONTROL STATEMENTS (SWITCH CASE)

- Switch case

```
switch statement; expression {
  case <value1>: …
  case <value2>: …
  default: …
}
```

- Switch with multiple expressions in case

```
switch statement; expression {
  case <value1>, <value2>, <value3>: …
  case <value4>: …
  default: …
}
```

- Expressionless switch

```
switch statement; {
  case expression1: …
  case expression2: …
  default: …
}
```

CONTROL STATEMENTS (FOR LOOP)

- A **for loop** is used for iterating over a sequence (that is either a slice, an array, a map, or a string.

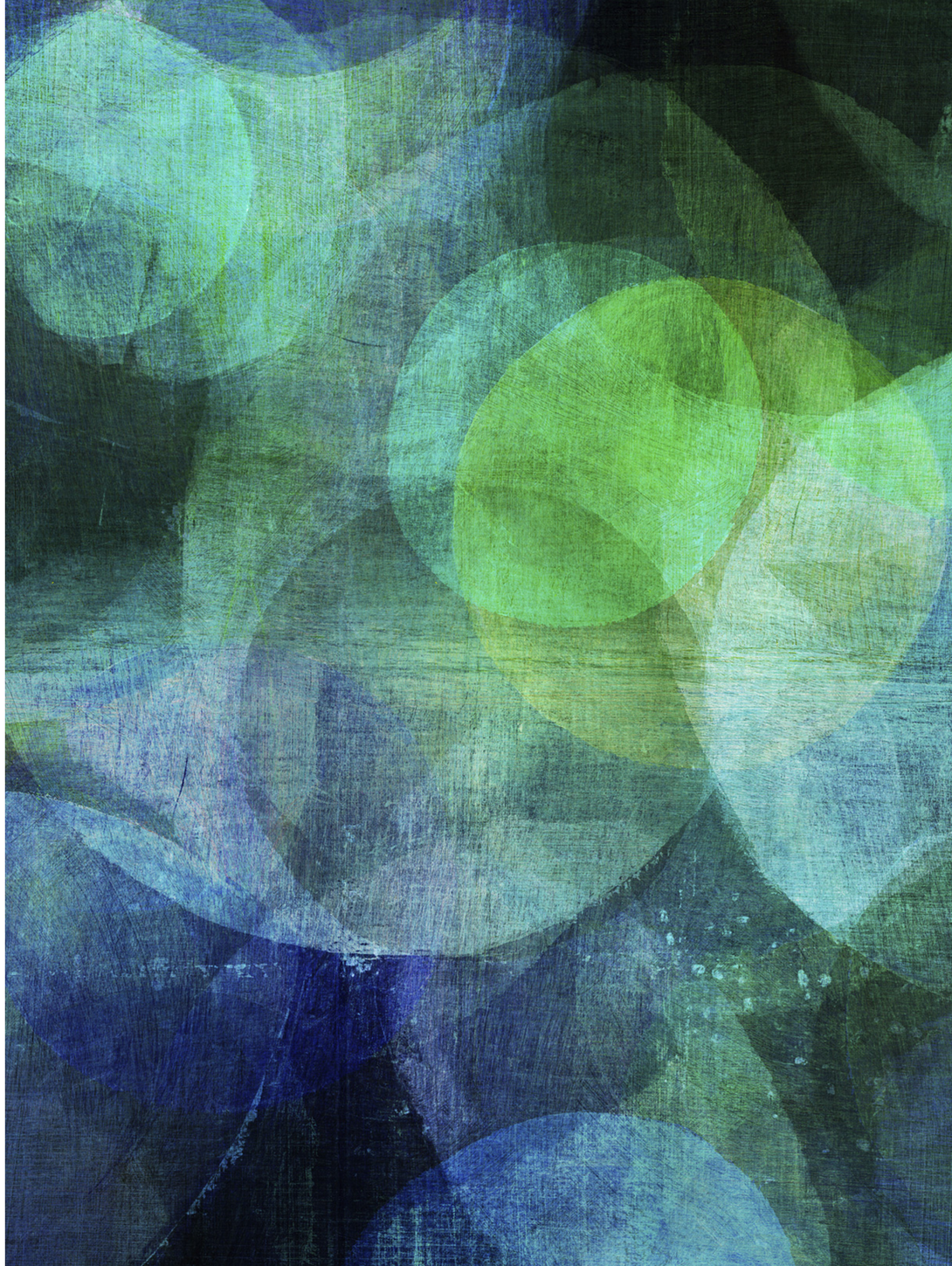
```
for i:=0; i<3; i++ { … }

for { … }

for i<5 { … }

for k := 1; ; k++ { … }

for i, v := range items { … } //Iterate over Slice/Array/String

for _, v := range items { … } //Iterate over Slice/Array/String

for k, v := range m { … } //Iterate over Map

j := 0
for range strSlice { //Iterate over Slice/String/Channel
    fmt.Println(strSlice[j])
    j++
}

for i := 0; i<2; i++ {
  for j := 0; j<2; j++ { … }
}
```

PRACTICE & REFERENCE (BASIC GOLANG):

- https://tour.golang.org/basics/1
- https://golangbyexample.com/workspace-hello-world-golang/
- https://www.slideshare.net/suelengc/go-lang-52138194
- https://www.slideshare.net/fmamud/ftd-golang
- https://www.golangprograms.com/go-language.html

Advanced:

1. Interfaces
2. Pointers
3. Error handling
4. Defer
5. Panic and Recover
6. Go Server
7. Go Testing
8. OOP



INTERFACE

- An interface is an abstract concept which enables **polymorphism** in Go. A variable of that interface can hold the value that implements the type. Type assertion is used to get the underlying concrete value.
- An Interface is an abstract type.

```
var a interface{}
```

- Interface describes all the methods of a method set and provides the signatures for each method. Interface can be used as a custom type that is used to specify a set of one or more method signatures. An interface is implemented when the type has implemented the functions of the interface.
- An interfaces act as a blueprint for method sets, they must be implemented before being used. Type that satisfies an interface is said to implement it.
- Interface lets you use duck typing in golang. *"If it walks like a duck and quack like a duck then it must be duck"*
- Interface is abstract, so you are not allowed to create an instance of the interface.

```
type Person interface {
   greet() string
}
```

INTERFACE

```
import "fmt"

type Person interface {
	greet() string
}
type Human struct {
	Name string
}
func (h *Human) greet() string {
	return "Hi, I am " + h.Name
}
func isAPerson(h Person) {
	fmt.Println(h.greet())
}

func main() {
	var a = Human{"John"}
	fmt.Println(a.greet()) // Hi, I am John

	var b = Human{"Jai"}
	var p Person = &b
	fmt.Println(p.greet()) // Hi, I am Jai

	// below function will only work
	// if a is also a person.
	// Here we can see polymorphism in action.
	isAPerson(&a) // Hi, I am John
}
```

TYPE SWITCH

- Type switch with single expression in case:

```
var value interface{} = "GeeksforGeeks"
switch newValue := value; t := newValue.(type) {
   case int: …
   case string: …
   default: …
}
```

- Type switch with multiple expression in case:

```
switch t := value.(type) {
   case int, int64: …
   case string: …
   default: …
}
```

POINTERS

- Pointers is a variable which is used to store the memory address of another variable. The variable can be of basic type or complex type.
- ***T** is the type of the pointer variable which points to a value of type **T**. The **&** operator is used to get the address of a variable.
- The zero value of a pointer is **`nil`**.
- Go also provides a handy function `new` to create pointers. The `new` function takes a type as an argument and returns a pointer to a newly allocated zero value of the type passed as argument.
- Dereferencing a pointer means accessing the value of the variable to which the pointer points. `*a` is the syntax to deference a.
- We can pass a pointer variable which holds the address of variable to the function. If the value of the pointer variable is changed using dereference then the value of original variable is modified too.
- Function can return a pointer to variable.

```
var x int = 5748
var p *int
p = &x
*p = 8234
```

ERROR HANDLING

- Errors are represented using the built-in **`error`** type.
- Just like any other built-in type such as int, float64, ... **error** values can be stored in **variables**, passed as **parameters** to functions, **returned** from functions, and so on.
- The idiomatic way of handling errors in Go is to compare the returned error to `nil`. A nil value indicates that no error has occurred and a non-nil value indicates the presence of an error.
- The **`errors.New`** function can be used to return a user-defined `error` with custom message.

ERROR HANDLING

```
package main
import (
    "errors"
    "fmt"
)
func Hello(name string) (string, error) {
    if name == "" {
        return "", errors.New("empty name”)
    }
    message := fmt.Sprintf("Hi, %v. Welcome!", name)
    return message, nil
}

func main() {
    message, err := greetings.Hello("")
    if err != nil {
        fmt.Println("Error:",err)
    }
    fmt.Println(message)

    message, err = greetings.Hello("Go")
    if err != nil {
        fmt.Println("Error:",err)
    }
    fmt.Println(message)
}
```

DEFER

- A **defer statement** pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns.
- Defer is commonly used to simplify functions that perform various clean-up actions.
- Defer statements allow us to think about closing each file right after opening it, guaranteeing that, regardless of the number of return statements in the function, the files *will* be closed.
- The behaviour of defer statements is straightforward and predictable. There are three simple rules:
  - A deferred function's arguments are evaluated when the defer statement is evaluated.

```
i := 0
defer fmt.Println(i)
i++
```

  - Deferred function calls are executed in Last In First Out order after the surrounding function returns.

```
for i := 0; i < 4; i++ {
    defer fmt.Print(i)
}
```

  - Deferred functions may read and assign to the returning function's named return values.

```
func c() (i int) {
    defer func() { i++ }()
    return 1
}
```

PANIC & RECOVER

- **Panic** is a built-in function that stops the ordinary flow of control and begins *panicking*. Panic is meant to exit from a program in abnormal conditions.
- Panic can occur in a program in two ways:
  - Runtime error in the program.
  - By calling the panic function explicitly. This can be called by the programmer when the program cannot continue and it has to exit.
- Runtime error in the program can happen in below cases
  - Out of bounds array access
  - Calling a function on a nil pointer
  - Sending on a closed channel
  - Incorrect type assertion
- **Recover** is a built-in function that regains control of a panicking goroutine. Recover is only useful inside deferred functions.
- During normal execution, a call to recover will return nil and have no other effect. If the current goroutine is panicking, a call to recover will capture the value given to panic and resume normal execution.
- **defer** function is the only function that is called after the **panic**. So it makes sense to put the **recover** function in the **defer** function only. If the **recover** function is not within the defer function then it will not stop **panic**.

GO SERVER

- Go is a great language for creating simple yet efficient web servers and web services. It provides a built-in HTTP package that contains utilities for quickly creating a web or file server.
- HTTP server is basically a program running on a machine. It listens and responds to HTTP requests on its IP address with a particular port.
- **net** package contains http package that provides both HTTP client (to make http requests) and HTTP server (listens to http requests) implementations.


```
package main

import (
        “io"
        "log"
        "net/http"
)

func main() {
        // Set routing rules
        http.HandleFunc("/", Home)
        //Use the default DefaultServeMux.
        err := http.ListenAndServe(":8080", nil)
        if err != nil {
                log.Fatal(err)
        }
}

func Home(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "version 1")
}
```

GO SERVER

```
package main

import (
   "net/http"
)

func main() {

   //Handling the /v1/teachers
   http.HandleFunc("/v1/teachers", teacherHandler)

   //Handling the /v1/students
   sHandler := studentHandler{}
   http.Handle("/v1/students", sHandler)

   http.ListenAndServe(":8080", nil)
}

func teacherHandler(res http.ResponseWriter, req *http.Request) {
   data := []byte("V1 of teacher's called")
   res.WriteHeader(200)
   res.Write(data)
}

type studentHandler struct{}

func (h studentHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
   data := []byte("V1 of student's called")
   res.WriteHeader(200)
   res.Write(data)
}
```

GO SERVER

- **Request** – it defines the request parameters i.e, Method, Api Signature, request headers, body, query params etc
- **Response** – defines the response parameters i.e, Status code, response body, headers
- **Pair of API signature and its handler** – Each API signature corresponds to a handler. You can think of handler as a function which is invoked when a request is made for that particular API signature. The **mux** registers these pairs of API signature and its handler
- **Route** – It acts as a router. Depending upon API signature of the request, it routes the request to the registered handler for that API signature. The handler will handle that incoming request and provide the response . For eg an API call with **"/v2/teachers"** might be handled by a different function and API call with **"/v2/students"** might be handled by some other function. So basically based upon API signature(and also request method sometimes) , it decides which handler to invoke.
- **Listener** – It runs on the machine, which listens to a particular port. Whenever it receives the request on that port it forwards the request to the **router**.

TESTING

- Package testing provides support for automated testing of Go packages.
- It is intended to be used in concert with the "go test" command, which automates execution of any function of the form ***func TestXxx(t *testing.T)*** where Xxx does not start with a lowercase letter.
- To create a test file, we should create “<file>_test.go” in the same folder where the code that needs to be tested already lies.
- Golang mocks are widely-used UT technique to examine the code interaction with cache, more specifically — how values are inputted and outputted from there.
- Benchmark tests provide you with reports on software performance.

OOP

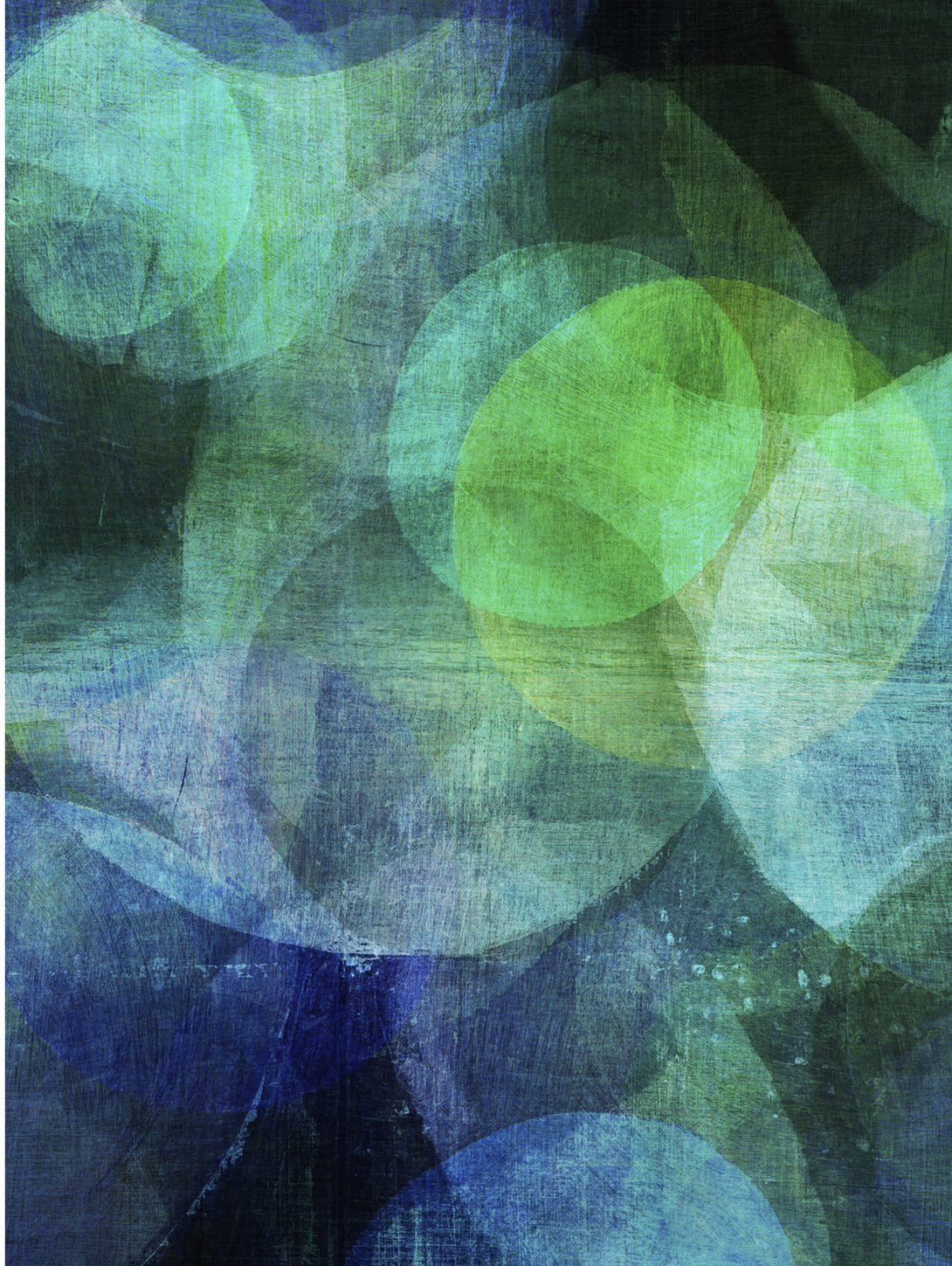
- Object-oriented programming is a programming paradigm which uses the idea of “objects” to represent data and methods. Go does not strictly support object orientation but is a lightweight object Oriented language.
- **Encapsulation:** Data hiding in contrast to other Object Oriented languages where there are four or more access-levels, Go simplifies this to only two levels:
  - *package scope*: object is only known in its package, it starts with a lowercase letter
  - *exported*: object is visible outside of its package because it starts with an uppercase letter. A type can only have methods defined in its package.
- **Inheritance:** Inheritance via composition that is embedding of 1 (or more) type(s) with the desired behaviour (fields and methods); multiple Inheritance is possible through embedding various type. One must use a struct to achieve Inheritance in Golang.
- **Polymorphism:** Interfaces elegantly provide polymorphism: by accepting an interface, you declare to accept any kind of object satisfying that interface. Types and interfaces are loosely coupled; again, multiple Inheritance is possible through implementing various interfaces.

PRACTICE:

- https://tour.golang.org/methods/1
- https://golangbyexample.com/golang-comprehensive-tutorial/
- https://github.com/uber-go/guide/blob/master/style.md
- https://www.youtube.com/watch?v=YS4e4q9oBaU
- https://www.youtube.com/watch?v=NU_1StN5Tkk
- https://www.youtube.com/watch?v=zzAdEt3xZ1M

Concurrency:

1. Concurrency
2. Goroutines
3. Channels
4. Select
5. WaitGroups
6. Mutex



CONCURRENCY

- Concurrency is the capability to deal with lots of things at once.
- Go is a concurrent language and not a parallel one.
- Concurrency is an inherent part of the Go programming language. Concurrency is handled in Go using Goroutines and channels.
- Example is a OS running on a single core processor machine:
  - A CPU core can handle one thing at a time.
  - When we talk about concurrency, we are doing one thing at a time but we divide CPU Time among things that need to be processed. Hence we get a sensation of multiple things happening at the same time in in reality, only one thing is happening at a time.
  - A single core processor pretty much divide the workload based on the priority of each task, for example, while page scrolling, listening to music may have a low priority, hence sometimes your music stops because of low internet speed but you can still scroll the page.

GOROUTINES

- A goroutine is a function or method that is capable of running concurrently with other functions.
- Goroutines can be thought of as lightweight threads. Goroutines are extremely cheap when compared to threads. They are only a few kb in stack size and the stack can grow and shrink according to the needs of the application whereas in the case of threads the stack size has to be specified and is fixed.
- A *goroutine* is a lightweight thread managed by the Go runtime.
- The cost of creating a Goroutine is tiny when compared to a thread. Hence it's common for Go applications to have thousands of Goroutines running concurrently.
- The Goroutines are multiplexed to a fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines.

GOROUTINES

- When a new Goroutine is started, the goroutine call returns immediately. Unlike functions, the control does not wait for the Goroutine to finish executing. The control returns immediately to the next line of code after the Goroutine call and any return values from the Goroutine are ignored.
- Goroutines don't have parents or children. When you start a goroutine it just executes alongside all other running goroutines. Each goroutine exits only when its function returns.
- The main Goroutine should be running for any other Goroutines to run. If the main Goroutine terminates then the program will be terminated and no other Goroutine will run.
- Prefix the function or method call with the keyword `go` and you will have a new Goroutine running concurrently.


```
func hello() {
   fmt.Println("Hello world goroutine”)
}
func main() {
   go hello()
   fmt.Println("main function")
   time.Sleep(1 * time.Second)
}
```

CHANNELS

- Famous Golang Quote: “Do not communicate by sharing memory instead share memory by communicating.”
- Channel is a data type in Go which provides synchronisation and communication between goroutines.
- Goroutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using Goroutines.
- A *race condition* is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly.
- Channels can be thought of as a pipe using which Goroutines communicate. channel default property is bidirectional, which means one goroutine can send and another can receive the request at one time, so in very simple we can say that channel perform two things one is sending and another receiving the data.

CHANNELS

- **Types of Channel:**
  - **Unbuffered channel:** A channel that can hold a single piece of data, which has to be consumed before pushing other data. Our main goroutine will get blocked when we add data into the channel.
  - **Buffered channel:** In a buffered channel, we specify the data capacity of a channel. The syntax is very simple. c := make(chan int, 10) the second argument in the make function is the capacity of a channel. So, we can put up to ten elements in a channel. When the capacity is full, then that channel would get blocked so that the receiver goroutine can start consuming it.
- Ways to initialise a channel:

```
var c chan string
```

or

```
c := make(chan string)
```

or

```
c := make(chan int, 10)
```

- Send to channel:

```
c <- “ping"
func pinger(c chan<- string)
```

- Receive from channel:

```
msg := <- c
func printer(c <-chan string)
```

SELECT

- Select is similar to switch statement, the difference being that in select each of the case statement waits for a send or receive operation from a channel.
- Select statement will wait until send or receive operation is completed on any one of the case statements. It is different from the switch statement in the way that each of the case statements will either send or receive operation on a channel whereas in switch each of the case statements is an expression. So a select statement lets you wait on multiple send and receive operations from different channels.
- Two important points to note about set a statement is
  - The select blocks until any of the case statements are ready.
  - If multiple case statements are ready than it selects one at random and proceeds.
- Syntax:

```
select {
   case msg1 := <-ch1:
       fmt.Println(“Message 1:”, msg1)
   case msg2 := <-ch2:
      fmt.Println“Message 2:”, (msg2)
   case <- time.After(time.Second):
     fmt.Println(“timeout")
}
```

WAIT GROUP

- **WaitGroup** is actually a type of counter which blocks the execution of function (or might say A goroutine) until **its internal counter become 0**.
- WaitGroup exports 3 methods:
  - **Add(int)**: It increases WaitGroup counter by given integer value.
  - **Done()**: It decreases WaitGroup counter by 1, we will use it to indicate termination of a goroutine.
  - **Wait()**: It Blocks the execution until it's internal counter becomes 0.
- Syntax:

```
func runner1(wg *sync.WaitGroup) {
    defer wg.Done() // This decreases counter by 1
    fmt.Print("\nI am first runner")
}
func runner2(wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Print("\nI am second runner")
}
func execute() {
    wg := new(sync.WaitGroup)
    wg.Add(2)

    // We are increasing the counter by 2
    // because we have 2 goroutines
    go runner1(wg)
    go runner2(wg)

    // This Blocks the execution
    // until its counter become 0
    wg.Wait()
}
```

MUTEX

- Mutex is short for mutual exclusion. Mutexes keep track of which thread has access to a variable at any given time.
- A Mutex is a method used as a locking mechanism to ensure that only one Goroutine is accessing the critical section of code at any point of time. This is done to prevent race conditions from happening.
- If one Goroutine already has the lock and if a new Goroutine is trying to get the lock, then the new Goroutine will be stopped until the mutex is unlocked.
- Mutex is used to avoid the **concurrent read/write problem**. This occurs when one thread is **writing** to a variable while another variable is concurrently **reading** from that same variable. The program will panic because the reader could be reading bad data that is being mutated in place.
- Mutex exports 2 methods:
  - **Lock()**: Acquire lock
  - **Unlock()**: Release lock

MUTEX

- Example:

```
package main
import (
   “fmt”
   “sync”
)

func f(v *int, wg *sync.WaitGroup, m *sync.Mutex) {
    // acquire lock
    m.Lock()
    // do operation
    *v++
    // release lock
    m.Unlock()
    wg.Done()
}

func main() {
    var wg sync.WaitGroup
    // declare mutex
    var m sync.Mutex
    var v int = 0
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go f(&v, &wg, &m)
    }
    wg.Wait()
    fmt.Println("Finished", v)
}
```

PRACTICE:

- https://tour.golang.org/concurrency/1
- https://www.youtube.com/watch?v=oV9rvDllKEg
- https://www.youtube.com/watch?v=f6kdp27TYZs

Thank You