GO LANGUAGE

Agenda

01 INTRODUCTION TO GO LANG, DATA STRUCTURES, FUNCTIONS, POINTERS

02 STRUCTURES, INTERFACES, PACKAGES, GO ROUTINES, CHANNELS, CONCURRENCY PATTERNS

03 CORE PACKAGES, BUILD REST API's USING GO, TESTING

Session-01
Installation Setup, Introduction & Basics

Why Go

- Go, or Golang, is an open source programming language. It's statically typed and produces compiled machine code binaries. Developers say that Google's Go language is the C for the twenty-first century when it comes to syntax. However, this new programming language includes tooling that allows you to safely use memory, manage objects, collect garbage, and provide static (or strict) typing along with concurrency.

- Ease of use together with state-of-the-art productivity

- High-level efficiency along with static typing

- Advanced performance for networking and the full use of multi-core power

Spend less time and money to develop an app

You don't need a huge teck stack if you're using Go for your project. Apps created in Go actually compile to native machine code and don't need any interpreter or virtual machine. This also means that Go apps will work faster and won't require the warming up we've just mentioned.

GO

Use Go for a range of apps

Go is a really flexible language, able to solve a lot of problems. You can use it for system and network programming, big data, machine learning, audio and video editing, and more.

GO

Get more performance and a wider audience for your app

Similarly to C or C++, Go is a compiled language and doesn't require any interpretation. Correspondingly, the absence of an interpreter frees up power and gives a Go-built app way more performance, which will surely be appreciated by users. Moreover, Go knows how to properly manage allocated memory.

Worry less about the app crashing

Go was created to use the full potential of multiple cores. Moreover, the language can properly use all the processor resources, so it's perfect for running an app in the background as a single process.

This is possible thanks to goroutines, which are used instead of threads and require much less RAM due to their non-system thread nature. This is why the risk of a Go app crashing due to lack of memory is lower.

INSTALLATION OF GO IN UBUNTU

- sudo apt-get update

- sudo apt-get -y upgrade

- wget https://dl.google.com/go/go1.13.3.linux-amd64.tar.gz

- sudo tar -xvf go1.13.3.linux-amd64.tar.gz

- sudo mv go /usr/local

GO

SETUP GO ENVIRONMENT

◻ You need to set 3 environment variables as GOROOT, GOPATH and PATH

◻ **GOROOT** is the location where Go package is installed on your system

◻ **echo 'export GOROOT=/usr/local/go'>>~/.profile**

◻ **GOPATH** is the location of your work directory. For example my project

directory is **$Home/Projects/Proj1**.

◻ **echo 'export GOPATH=$HOME/Projects/Proj1'>>~/.profile**

◻ Now set the **PATH** variable to access go binary system wide.

◻ **echo 'export PATH=$GOPATH/bin:$GOROOT/bin:$PATH'>>~/.profil**e

◻ **source ~/.profile**

VERIFY INSTALLATION

☐   To check the version:

  go version

☐   To verify the configured environment variables:

  go env

GO

BASIC STRUCTURE OF GO PROGRAM

It consists of:

- Package Declaration

- Import Packages

- Variables

- Statements and Expressions

- Functions

- Comments

GO

EXAMPLE PROGRAM

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

GO

HOW TO RUN A PROGRAM

- We have saved the program as Hello.go.

- To run the program, open the command prompt and type:

  go run Hello.go

GO

DATA TYPES

- Bool
- String
- Int
- Int8
- Int16
- Int32
- Int64
- Float32
- Float64

- Uint
- Uint8
- Uint16
- Uint32
- Complex
- Complex64
- Complex128
- uintptr

GO

GO LANGUAGE KEYWORDS

Break	Default	Func	Interface	Select
Case	Defer	Go	Map	Struct
Chan	Else	Goto	Package	Switch
Const	Fallthrough	If	Range	Type
continue	For	Import	Return	var

GO

VARIABLES

The general form for declaring a variable uses the keyword var

Syntax:-
    var identifier type

Example:-
    var a int
    var b bool
    var str string

GO

VARIABLES DECLARATION METHODS

01
```
var i int
var s string
i=10
s="hello world"
```

02
```
var i int = 10
var s string ="hello world"
```

03
```
var i = 10
var s ="hello world"
```

GO

SHORT VARIABLES DECLARATION

:= is the short variable assignment operator to declare the variable.

No need to use the var keyword or declare the variable type while short variable declaration.

country := "India"

MULTIPLE VARIABLES DECLARATION

Golang allows you to assign values to multiple variables in one line.

Examples:-
- **var fname, lname string = "John", "Doe"**
- **m, n, o := 1, 2, 3**
- **item, price := "Mobile", 2000**

GO

CONSTANTS DECLARATION METHODS

01

Explicit Type

const PRODUCT string = "India"

02

Implicit Type

const PRICE = 500

03

Multiple Variables

const (
    PRODUCT = "Mobile"
    QUANTITY = 50
    PRICE   = 50.50
    STOCK = true
)

GO

FILES AND FOLDERS

The most important package that allows us to manipulate files and directories as entities is the **os** package.

The io package has the **io.Reader** interface to reads and transfers data from a source into a stream of bytes.

The **io.Writer** interface reads data from a provided stream of bytes and writes it as output to a target resource.

GO

Session-02

Control Structures & Data Strucutres

CONTROL STRUCTURES

- If
- If-else
- For
- For range
- Switch

Syntax:-
if condition {
    // code to be executed
if condition is true
}

Example:-
var s = "Japan"
x := true
if x {
    fmt.Println(s)
}

Syntax:-
if var declaration; condition {
    // code to be executed if
condition is true
}

Example:-
if x := 100; x == 100 {
    fmt.Println("Germany")
}

GO

CONTROL STRUCTURES

If

If-else

For

For range

Switch

Syntax:-

if condition {
    // code to be executed if condition is true
} else {
    // code to be executed if condition is false
}

Example:-

x := 100
if x == 100 {
  fmt.Println("Japan")
} else {
  fmt.Println("Canada")
}

GO

CONTROL STRUCTURES

If

If-else

For

For range

Switch

Examples:-

```
k := 1
for ; k <= 10; k++ {
 fmt.Println(k)
}
k = 1
for k <= 10 {
 fmt.Println(k)
 k++
}
for k := 1; ; k++ {
 fmt.Println(k)
 if k == 10 {
 break
 }
}
```

GO

CONTROL STRUCTURES

If

If-else

For

For range

Switch

Syntax:-
for ix, val := range coll { }

Example:-
nums := []int{2, 3, 4}
  sum := 0
  for _, value := range nums {// "_ " is to ignore the index
    sum += value
  }
  fmt.Println("sum:", sum)

GO

CONTROL STRUCTURES

If

If-else

For

For range

Switch

Syntax:-
switch var1 {
case val1:
.....
case val2
.....
default:
.....
}

Example:-   var input int
fmt.Scanln(&input)
switch (input) {
case 10:
    fmt.Print("the value is 10")
case 20:
    fmt.Print("the value is 20")
case 30:
    fmt.Print("the value is 30")
case 40:
    fmt.Print("the value is 40")
default:
    fmt.Print(" It is not 10,20,30,40 ")
}

GO

ARRAYS

 In Go, an array is a homogeneous data structure (Fix type) and has a fixed length.

 The type can be anything like integers, string or self-defined type.

 Syntax:
   var identifier [len]type
 Example:
   var arr_name [5]int

MULTI DIMENSIONAL ARRAYS

- Multi dimensional arrays is simply a list of one-dimensional arrays.

- **Syntax:**
  var arrayName [x][y] variable_type

- **Example:**
  a = [3][4]int

GO

INITIALIZING & ACCESSING TWO DIMENSIONAL ARRAYS

- **Initializing Two Dimensional Arrays:-**

```
a = [2][3]int{
    {2, 4, 6} ,   /*  initializers for row indexed by 0 */
    {8, 10, 12} ,   /*  initializers for row indexed by 1 */
}
```

- **Accessing Two Dimesnional Arrays:-**

```
int val = a[1][2]
```

GO

EXAMPLE

```go
/* an array with 3 rows and 3 columns*/
  var a = [3][3]int{ {1,2,3}, {4,5,6}, {7,8,9}}
  var i, j int
  /* output each array element's value */
  for  i = 0; i < 3; i++ {
    for j = 0; j < 3; j++ {
      fmt.Print(a[i][j] )
    }
    fmt.Println()
  }
```

GO

SLICES

- In Go, a slice is a flexible and extensible data structure to implement and manage collections of data.

- A slice is a segment of dynamic arrays that can grow and shrink.

- The only difference between this and an array is the missing length between the brackets. In this case, variable has been created with a length of 0.

SLICES

- **Syntax:**
  var identifier [ ]type

- **Example:**
  var slice_name [ ]int

GO

SLICES

- Slice is a dynamically-sized, segmented view of an underlying array.

- This segment can be the entire array or a subset of an array.

- We define the subset of an array by indicating the start and end index.

- Slices provide a dynamic window onto the underlying array.

- **Example:-**

```
odd := [6]int{2, 4, 6, 8, 10, 12}
var s []int = odd[1:4]
```

GO

SLICES

- Slice is like reference to an array.

- Slice does not store any data.

- If we change the elements of an array, it will also modify the underlying array.

- If other slice is referencing the same underlying array, their value will also be changed.

- Slice literal is like an array literal without any length.

- In slice, we can omit the lower bond or the upper bonds. Zero is the def... value of the lower or the upper bond.

SLICES

- A slice has length and capacity.

- The length is the number of stored elements and the capacity is the number of elements of the underlying array counting from the beginning of the slice.

- To get the length, we use len(slice) function and to get the capacity, we use cap(slice) function.

- We can also create slice with the help of make function. The make function creates a zero sized array and returns slice of the array.

   Note: Refer slices examples

GO

```go
arr := [7]string{"This", "is", "the", "tutorial",
                              "of", "Go", "language"}

    // Display array
    fmt.Println("Array:", arr)

    // Creating a slice
    myslice := arr[1:6]

    // Display slice
    fmt.Println("Slice:", myslice)

    // Display length of the slice
    fmt.Printf("Length of the slice: %d",
len(myslice))

    // Display the capacity of the slice
    fmt.Printf("\nCapacity of the slice: %d",
cap(myslice))
```
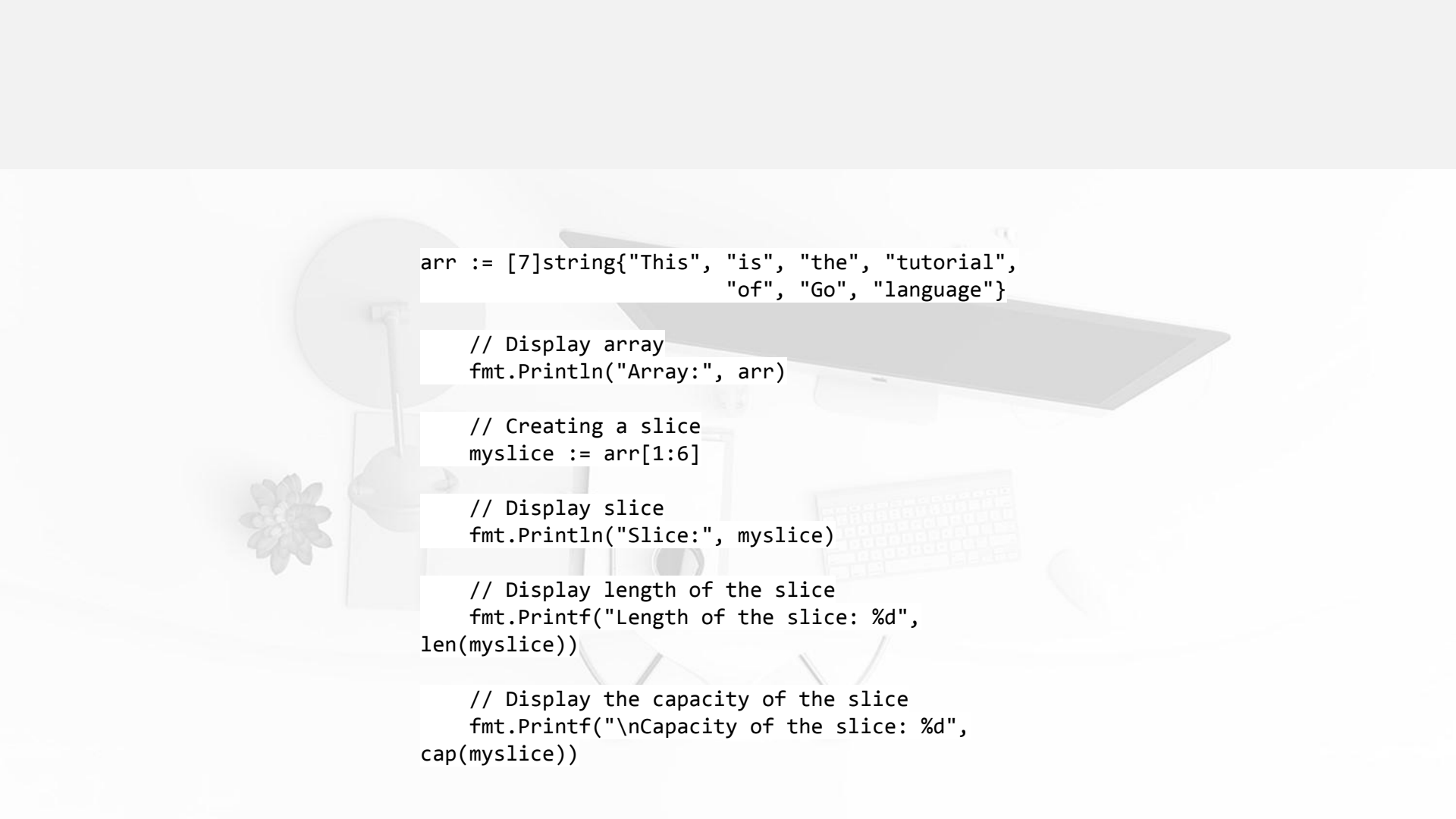
MAPS

☐ A map is an unordered collection of key-value pairs. Also known as an associative array, a hash table or a dictionary, maps are used to look up a value by its associated key.

Syntax:
    var identifier map[string]int
Example1:
    var x map[string]int
    x["key"] = 10
    fmt.Println(x)

GO

GO MAP OPERATIONS

 Update and insert operation are similar in go map.

 If the map does not contain the provided key the insert operation will takes place and if the key is present in the map then update operation takes place.

 You can delete an element in Go Map using delete() function.
Syntax:
  delete(map, key)

 Go Map Retrieve Element
Syntax:
  elem = m[key]
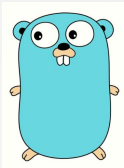
GO MAP OF STRUCT

In Go, Maps are like struct, but it requires keys

```go
package main
import "fmt"
type Vertex struct {
    Latitude, Longitude float64
}
var m = map[string]Vertex{
    "JavaTpoint": Vertex{    40.68433, -74.39967,  },
    "SSS-IT": Vertex{    37.42202, -122.08408,  },
}
func main() {
    fmt.Println(m)
}
```

Session-03
Functions, Pointers & Error Handling

FUNCTIONS

- A function is a group of statements that exist within a program for the purpose of performing a specific task. At a high level, a function takes an input and returns an output.

- Function allows you to extract commonly used block of code into a single component.

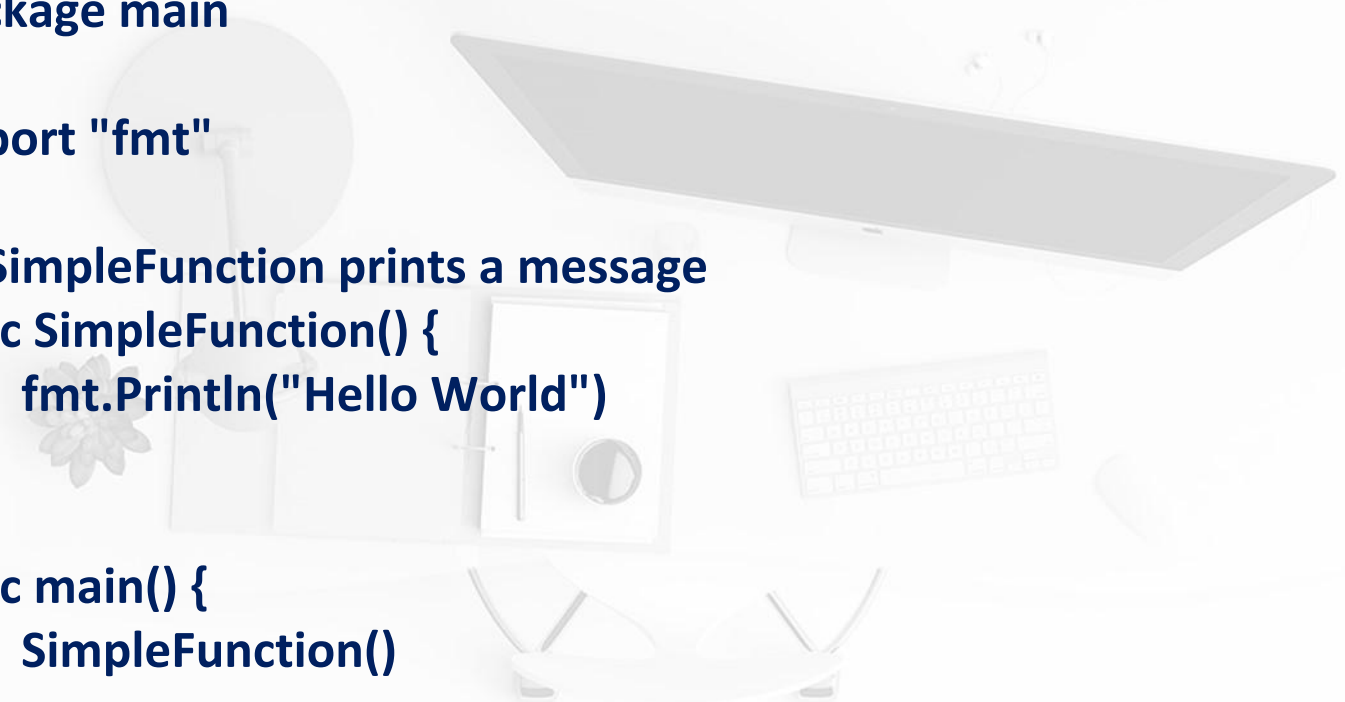- The single most popular Go function is **main()**, which is used in every independent Go program.

Example

```go
package main

import "fmt"

// SimpleFunction prints a message
func SimpleFunction() {
    fmt.Println("Hello World")
}

func main() {
    SimpleFunction()
}
```

Creating a Function

- A declaration begins with the func keyword, followed by the name you want the function to have, a pair of parentheses (), and then a block containing the function's code.

- The following example has a function with the name **SimpleFunction**.

- It takes no parameter and returns no values.

CLOSURE

- Here, we create an anonymous function which acts as a function closure.

- A function which has no name is called anonymous function.

- A closure is a function which refers reference variable from outside its body.

- The function may access and assign to the referenced variables.

GO

CLOSURE EXAMPLE

```go
package main
import (
  "fmt"
)
func main() {
  number := 10
    squareNum := func() (int){
    number *= number
    return number
  }
  fmt.Println(squareNum())
  fmt.Println(squareNum())
}
```

VARIADIC FUNCTIONS

 A variadic function is a function that accepts a variable number of arguments.

 In Golang, it is possible to pass a varying number of arguments of the same type as referenced in the function signature.

 To declare a variadic function, the type of the final parameter is preceded by an ellipsis, "...", which shows that the function may be called with any number of arguments of this type.

 This type of function is useful when you don't know the number of arguments you are passing to the function, the best example is built-in Println function of the fmt package which is a variadic function.

EXAMPLE

In below example, we are going to print s[0] the first and s[3] the forth, argument value passed to variadicExample() function.

```go
package main
import "fmt"

func main() {
    variadicExample("red", "blue", "green", "yellow")
}

func variadicExample(s ...string) {
    fmt.Println(s[0])
    fmt.Println(s[3])
}
```

ERROR HANDLING IN GO

☐ Go does not have an exception mechanism like try/catch in Java, we cannot throw an exception in Go.

☐ Go uses a different mechanism which is known as **_defer-panic-and-recover mechanism_**.

☐ Go handles simple errors for function, methods by returning an error object. The error object may be the only or the last return value. The error object is nil if there is no error in the function.

ERROR HANDLING IN GO

⬜ We should always check the error at the calling statement,
if we receive any of it or not.

⬜ We should never ignore errors, it may lead to program crashes.

⬜ The way go detect and report the error condition is:
   ❖ A function which can result in an error returns two variables: a value
      and an error-code which is nil in case of success, and != nil in case
      of an error-condition.
   ❖ The error is checked, after the function call . In case of an error
      (if error != nil), the execution of the actual function (or if necessary
      the entire program) is stopped.

ERROR HANDLING IN GO

- Go has predefined error interface type:

  type error interface {

      Error() string

  }

- We can define error type by using error.New from the error package and provide an appropriate error message like:

  err := errors.New("math - square root of negative number")

GO

EXAMPLE FOR ERROR HANDLING

```go
package main
import "errors"
import "fmt"
import "math"

func Sqrt(value float64) (float64, error) {
  if (value < 0) {
    return 0, errors.New("Math: negative number passed to Sqrt")
  }
  return math.Sqrt(value), nil
}
```

GO

EXAMPLE CONTINUATION

```go
func main() {
  result, err := Sqrt(-64)
  if err != nil {
    fmt.Println(err)
  } else {
    fmt.Println(result)
  }
  result, err = Sqrt(64)
  if err != nil {
    fmt.Println(err)
  } else {
    fmt.Println(result)
  }
}
```

GO

DEFER

- *Defer* statement defers the execution of a function until the surrounding function returns.

- They are generally used to execute necessary closing statements, for
example closing a file after you are done with it.

- Multiple defers are pushed into stack and executes in Last In First Out (LIFO) order.

- Defer generally used to clean up resources like a file, database connection etc.

PANIC

- ***Panic*** is similar to throwing an exception like other languages.

- Generally when a panic is called, then the normal execution flow stops immediately, but the deferred functions are executed normally.

- It is a built-in function in Golang.

GO

RECOVER

- *Recover* is another built-in function in go.

- It helps to regain the normal flow of execution after a panic.

- Generally, it used with a defer statement to recover panic in a goroutine.

- (Refer errors.go program)

GO

POINTERS

◻ **Pointers** in Go programming language or Golang is a variable which is used to store the memory address of another variable.

◻ You can also pass the pointers to the function like the variables.

◻ There are two ways to do this as follows:
  ✔ Create a pointer and simply pass it to the function
  ✔ Passing an address of the variable

IMPORTANT OPERATORS IN POINTERS

There are two important operators which we will use in pointers i.e.

- ***Operator** also termed as the dereferencing operator used to declare pointer variable and access the value stored in the address.

- **&Operator** termed as address operator used to returns the address of a variable or to access the address of a variable to a pointer.

GO

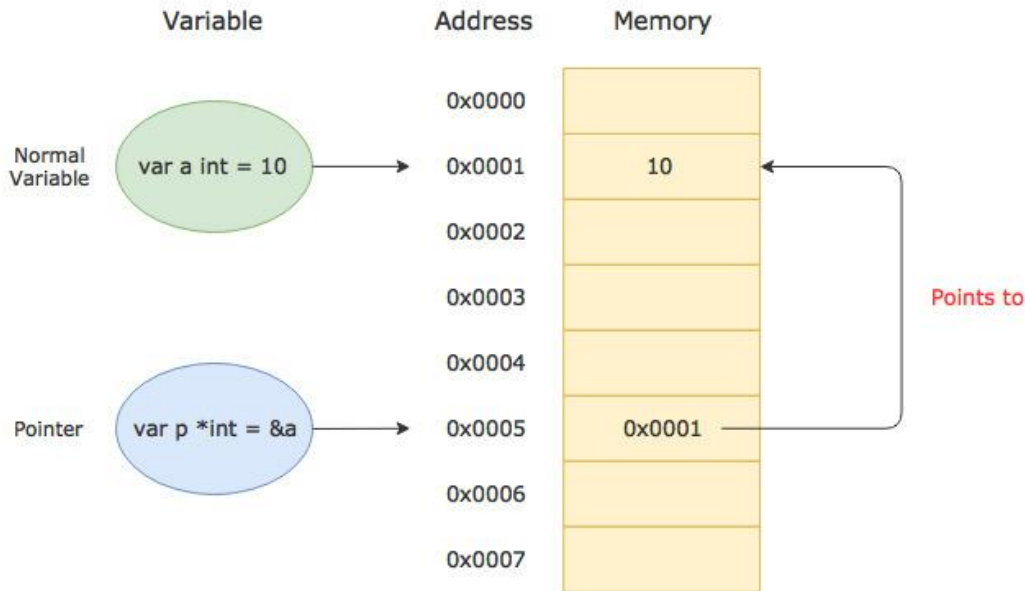# DECLARATION & INITIALIZATION OF POINTERS

Example:

var a int = 10

var p *int = &a

fmt.Println(a)

fmt.Println(*p)

GO

EXAMPLE PROGRAM

```go
package main
import "fmt"
func main() {

    i, j := 42, 2701

    p := &i         // point to i
    fmt.Println(*p) // read i through the pointer
    *p = 21         // set i through the pointer
    fmt.Println(i)  // see the new value of i

    p = &j          // point to j
    *p = *p / 37    // divide j through the pointer
    fmt.Println(j)  // see the new value of j
}
```

IMPORTANT POINTS

☐ The default value or zero-value of a pointer is always **nil**. Or you can say that an uninitialized pointer will always have a nil value.

```
// taking a pointer
var s *int

// displaying the result
fmt.Println("s = ", s)
```

Output:-
s = <nil>

IMPORTANT POINTS

☐ Declaration and initialization of the pointers can be done into a single line.
Example:
var s *int = &a

☐ You can also use the *shorthand (:=)* syntax to declare and initialize the pointer variables.

☐ The compiler will internally determine the variable is a pointer variable if we are passing the address of variable using *&(address)* operator to it.
Example:
y := 458
p := &y

GO

IMPORTANT POINTS

- If you are specifying the data type along with the pointer declaration then the pointer will be able to handle the memory address of that specified data type variable.

- For example, if you taking a pointer of string type then the address of the variable that you will give to a pointer will be only of string data type variable, not any other type.

GO

IMPORTANT POINTS

- To overcome the above mention problem you can use the Type Inference concept of var keyword.

- There is no need to specify the data type of during the declaration.

- The type of a pointer variable can also be determined by the compiler like a normal variable.

- Here we will not use the * operator.

- It will internally determine by the compiler as we are initializing the variable with the address of another variable.

GO

USE OF TYPE INFERENCE

```go
package main
import "fmt"
func main() {

    // using var keyword we are not defining any type with variable
    var y = 458

    // taking a pointer variable using  var keyword without specifying  the type
    var p = &y

    fmt.Println("Value stored in y = ", y)
    fmt.Println("Address of y = ", &y)
    fmt.Println("Value stored in pointer variable p = ", p)
}
```

Project-01
Design and develop a simple GO program

DAY-2

STRUCTURES

METHODS

INTERFACES

PACKAGES

GO ROUTINES

CHANNELS

CONCURRENCY PATTERNS

Session-04
Structures, Methods, Interfaces & Packages

STRUCTURES

- In Go, Struct can be used to create user-defined types.

- Struct is a composite type means it can have different properties and each property can have their own type and value.

- Struct can represent real-world entity with these properties.

- We can access a property data as a single entity.

- It is also valued types and can be constructed with the new() function.

SYNTAX

☐ The declaration starts with the keyword type, then a name for the new struct, and finally the keyword struct.

☐ Within the curly brackets, a series of data fields are specified with a name and a type.

```
type identifier struct{
    field1 data_type
    field2 data_type
    field3 data_type
    }
```

DECLARING A STRUCTURE

☐   For Example, an address has a name, street, city, state, Pincode.

☐   It makes sense to group these three properties into a single structure address as shown below.

```go
type Address struct {
    name string
    street string
    city string
    state string
    Pincode int
}
```

DEFINING A STRUCTURE

 Syntax:-

    var a Address

The above code creates a variable of a type Address which is by default set to zero. For a struct, zero means all the fields are set to their corresponding zero value. So the fields name, street, city, state are set to "", and Pincode is set to 0.

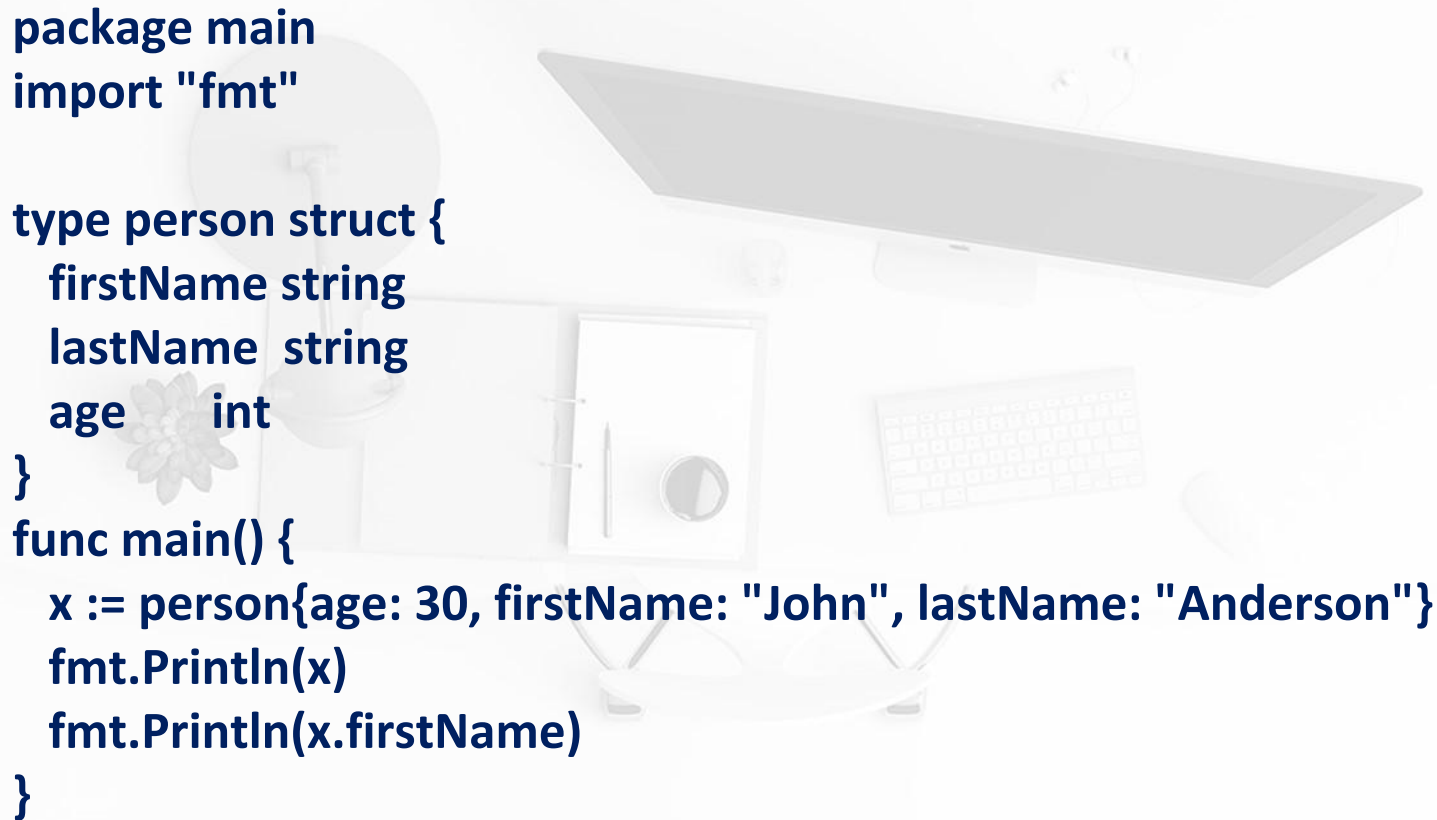 You can also initialize a variable of a struct type using a struct literal as shown below:

    var a = Address{"Akshay", "PremNagar", "Dehradun", "Uttarakhand", 252636}

EXAMPLE

```go
package main
import "fmt"

type person struct {
    firstName string
    lastName  string
    age       int
}
func main() {
    x := person{age: 30, firstName: "John", lastName: "Anderson"}
    fmt.Println(x)
    fmt.Println(x.firstName)
}
```

STRUCT INITIALISATION USING NEW KEYWORD

☐ An instance of a **struct** can also be created with the new keyword.

☐ It is then possible to assign data values to the data fields using dot notation.

```
type rectangle struct {
    length  int
    breadth int
    color   string
}
rect1 := new(rectangle)
(or)
var rect1 = new(rectangle)
```

STRUCT INITIALISATION USING POINTER ADDRESS OPERATOR(&)

- Creates an instance of rectangle struct by using a pointer address operator is denoted by **&** symbol.

```
var rect1 = &rectangle{10, 20, "Green"}
fmt.Println(rect1)
```

```
var rect2 = &rectangle{}
rect2.length = 10
rect2.color = "Red"
fmt.Println(rect2) // breadth skipped
```

NESTED STRUCTURES

☐ Struct can be nested by creating a Struct type using other Struct types as the type for the fields of Struct.

☐ Nesting one struct within another can be a useful way to model more complex structures.

```
type Salary struct {
    Basic, HRA, TA float64
}
type Employee struct {
    FirstName, LastName, Email string
    Age                        int
    MonthlySalary              []Salary
}
```

ADDING METHOD TO STRUCT TYPE

- You can also add methods to struct types using a method receiver.

- A method EmpInfo is added to the Employee struct

Example:-

```
func (e Employee) EmpInfo() string {
    fmt.Println(e.FirstName, e.LastName)
    fmt.Println(e.Age)
    fmt.Println(e.Email)
    for _, info := range e.MonthlySalary {
        fmt.Println("===================")
        fmt.Println(info.Basic)
        fmt.Println(info.HRA)
        fmt.Println(info.TA)
    }
    return "--------------------"
}
```
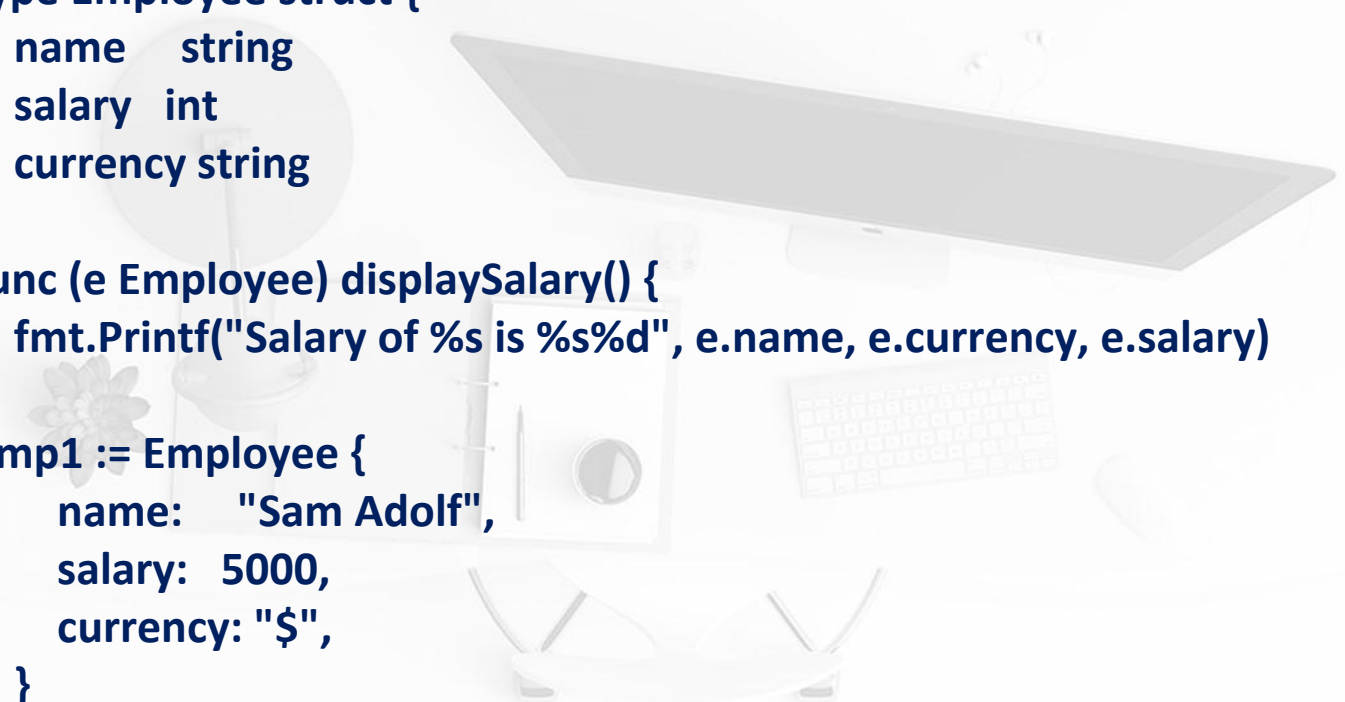
METHODS

- A method is just a function with a special receiver type between the func keyword and the method name. The receiver can either be a struct type or non-struct type.

- **Syntax:-**
  func (t Type) methodName(parameter list) {
  }

GO

EXAMPLE

```go
type Employee struct {
    name     string
    salary   int
    currency string
}
func (e Employee) displaySalary() {
    fmt.Printf("Salary of %s is %s%d", e.name, e.currency, e.salary)
}
emp1 := Employee {
    name:     "Sam Adolf",
    salary:   5000,
    currency: "$",
    }
emp1.displaySalary() //Calling displaySalary() method of Employee type
```

OUTPUT

- We have created a method displaySalary on Employee struct type.

- The displaySalary() method has access to the receiver e inside it.

- We are using the receiver e and printing the name, currency and salary of the employee.

- We have called the method using syntax emp1.displaySalary().

- This program prints Salary of Sam Adolf is $5000.

GO

METHODS vs FUNCTIONS

- Go is not a pure object-oriented programming language and it does not support classes.

- Hence methods on types are a way to achieve behavior similar to classes

- Methods allow a logical grouping of behavior related to a type similar to classes.

- In the above sample program, all behaviors related to the Employee type can be grouped by creating methods using Employee receiver type.

GO

METHODS vs FUNCTIONS

- Methods with the same name can be defined on different types whereas functions with the same names are not allowed.

- Let's assume that we have a Square and Circle structure.

- It's possible to define a method named Area on both Rectangle and Circle.

GO

METHODS vs FUNCTIONS

```go
type Rectangle struct {
    length int
    width  int
}
type Circle struct {
    radius float64
}
func (r Rectangle) Area() int {
    return r.length * r.width
}
func (c Circle) Area() float64 {
    return math.Pi * c.radius * c.radius
}

r := Rectangle{
    length: 10,
    width:  5,
}
fmt.Printf("Area of rectangle %d\n", r.Area())
c := Circle{
    radius: 12,
}
fmt.Printf("Area of circle %f", c.Area())
```

GO

INTERFACES

- In Go, an interface is a set of method signatures.

- When a type provides definition for all the methods in the interface, it is said to implement the interface

- Go has different approaches to implement the concepts of object-orientation.

- Go does not have classes and inheritance. Go fulfill these requirements through its powerful interface.

- Interfaces provide behavior to an object: if something can do this, then it can be used here.

INTERFACES

An interface defines a set of abstract methods and does not contain any variable.

Syntax:-
type interface_name interface {
       Method1(param_list) return_type
       Method2(param_list) return_type
       ...
}

INTERFACE INTERNAL REPRESENTATION

 An interface can be thought of as being represented internally by a tuple (type, value).

 Type is the underlying concrete type of the interface and value holds the value of the concrete type.

 (Refer example4.go in interfaces)

EMPTY INTERFACE

- An interface that has zero methods is called an empty interface.

- It is represented as interface{}.

- Since the empty interface has zero methods, all types implement the empty interface.

GO

TYPE ASSERTION

- Type assertion is used to extract the underlying value of the interface.

- i.(T) is the syntax which is used to get the underlying value of interface i whose concrete type is T.

GO

TYPE ASSERTION EXAMPLE

```go
package main
import (
    "fmt"
)

func assert(i interface{}) {
    s := i.(int) fmt.Println(s)
}
func main() {
    var s interface{} = 56
    assert(s)
}
```

- The concrete type of 's' is int.
- We use the syntax i.(int) to fetch the underlying int value of i.
- This program prints 56.

TYPE ASSERTION EXAMPLE WITH PANIC

```go
package main
import (
    "fmt"
)

func assert(i interface{}) {
    s := i.(int)
    fmt.Println(s)
}
func main() {
    var s interface{} = "Steven Paul"
    assert(s)
}
```

- In the above program we pass 's' of concrete type string to the assert function which tries to extract a int value from it.
- This program will panic with the message:
  panic: interface conversion: interface {} is string, not int

GO

SOLUTION

- To solve the above problem, we can use the syntax

   v, ok := i.(T)

- If the concrete type of i is T then v will have the underlying value of i and ok will be true.

- If the concrete type of i is not T then ok will be false and v will have the zero value of type T and the program will not panic.

GO

TYPE ASSERTION EXAMPLE WITHOUT PANIC

```go
package main
import (
    "fmt"
)
func assert(i interface{}) {
    v, ok := i.(int)
    fmt.Println(v, ok)
}
func main() {
    var s interface{} = 56
    assert(s)
    var i interface{} = "Steven Paul"
    assert(i)
}
```

- When Steven Paul is passed to the assert function, ok will be false since the concrete type of

  i is not int and v will have the value 0 which is the zero value of int. This program will print,
  56 true
  0 false

TYPE SWITCH

- A type switch is used to compare the concrete type of an interface against multiple types specified in various case statements.

- It is similar to switch case.

- The only difference being the cases specify types and not values as in normal switch.

   Syntax:-
   i.(type)

TYPE SWITCH

```go
func findType(i interface{}) {
    switch i.(type) {
    case string:
        fmt.Printf("I am a string and my value is %s\n", i.(string))
    case int:
        fmt.Printf("I am an int and my value is %d\n", i.(int))
    default:
        fmt.Printf("Unknown type\n")
    }
}
findType("Naveen")
findType(77)
findType(89.98)
```

GO

Session-05
Concurrency, Go Routines & Channels

CONCURRENCY

- Concurrency is the capability to deal with lots of things at once.

- When this browser is run in a single core processor, the processor will context switch between the two components of the browser.

- It might be downloading a file for some time and then it might switch to render the html of a user requested web page.

- This is know as concurrency. Concurrent processes start at different points of time and their execution cycles overlap.

- In this case the downloading and the rendering start at different points in time and their executions overlap.

PARALLELISM

☐ Parallelism is doing lots of things at the same time. It might sound similar to concurrency but it's actually different.

☐ Lets say the same browser is running on a multi core processor.

☐ In this case the file downloading component and the HTML rendering component might run simultaneously in different cores.

☐ This is known as parallelism.

GO

CONCURRENCY vs PARALLELISM

CONCURRENCY IN GO

- Concurrency is an inherent part of the Go programming language.

- Concurrency is handled in Go using Goroutines and channels.

- Concurrency in Golang is the ability for functions to run independent of each other.

GO

GOROUTINES

- Goroutines are functions or methods that run concurrently with other functions or methods

- Golang provides Goroutines as a way to handle operations concurrently.

- Goroutines can be thought of as light weight threads.

- The cost of creating a Goroutine is tiny when compared to a thread.

- Hence its common for Go applications to have thousands of Goroutines running concurrently.

GOROUTINES

☐ New goroutines are created by the go statement.

☐ To run a function as a goroutine, call that function prefixed with the go statement.

☐   Here is the example code block:

sum()
// A normal function call that executes sum synchronously and waits for completing it

go sum() // A goroutine that executes sum asynchronously and doesn't wait for completing it

GOROUTINES

 The go keyword makes the function call to return immediately, while the function starts running in the background as a goroutine and the rest of the program continues its execution.

 The main function of every Golang program is started using a goroutine, so every Golang program runs at least one goroutine.

ADVANTAGES OF GOROUTINES OVER 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 needs of the application whereas in the case of threads the stack size has to be specified and is fixed.

GO

ADVANTAGES OF GOROUTINES OVER THREADS

☐ The Goroutines are multiplexed to fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines. If any Goroutine in that thread blocks say waiting for user input, then another OS thread is created and the remaining Goroutines are moved to the new OS thread. All these are taken care by the runtime and we as programmers are abstracted from these intricate details and are given a clean API to work with concurrency.

ADVANTAGES OF GOROUTINES OVER THREADS

 Goroutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using Goroutines. Channels can be thought of as a pipe using which Goroutines communicate.

GO

CREATION OF A GOROUTINE

```go
package main

import (
    "fmt"
)

func hello() {
    fmt.Println("Hello world goroutine")
}
func main() {
    go hello()
    fmt.Println("main function")
}
```

- Our Goroutine did not run in this case.
- After the call to go hello, the control returned immediately to the next line of code without waiting for the hello goroutine to finish and printed main function.
- Then the main Goroutine terminated since there is no other code to execute and hence the hello Goroutine did not get a chance to run.

CREATION OF A GOROUTINE

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

- We have called the Sleep method of the time package which sleeps the go routine in which it is being executed.
- In this case the main goroutine is put to sleep for 1 second.
- Now the call to go hello() has enough time to execute before the main Goroutine terminates.
- This program first prints Hello world goroutine, waits for 1 second and then prints main function.

CHANNELS

- A channel is a medium through which a goroutine communicates with another goroutine and this communication is lock-free.

- Or in other words, a channel is a technique which allows to let one goroutine to send data to another goroutine.

- By default channel is bidirectional, means the goroutines can send or receive data through the same channel

CREATING A CHANNEL

- In Go language, a channel is created using chan keyword and it can only transfer data of the same type, different types of data are not allowed to transport from the same channel.

  Syntax:

  var Channel_name chan Type

- You can also create a channel using make() function using a shorthand declaration.

  Syntax:

  channel_name:= make(chan Type)

EXAMPLE

```go
package main

import "fmt"

func main() {
    var a chan int
    if a == nil {
        fmt.Println("channel a is nil, going to define it")
        a = make(chan int)
        fmt.Printf("Type of a is %T", a)
    }
}
```

Output:-

channel a is nil, going to define it
Type of a is chan int

CHANNEL OPERATIONS

- In Go language, channel work with two principal operations one is sending and another one is receiving, both the operations collectively known as communication.
- **<-** operator is used for communication.
- And the direction of <- operator indicates whether the data is received or send. In the channel, the send and receive operation block until another side is not ready by default.
- It allows goroutine to synchronize with each other without explicit locks or condition variables.

SEND OPERATION

- The send operation is used to send data from one goroutine to another goroutine with the help of a channel.
- Values like int, float64, and bool can safe and easy to send through a channel because they are copied so there is no risk of accidental concurrent access of the same value.
- Strings are also safe to transfer because they are immutable.
- But for sending pointers or reference like a slice, map, etc. through a channel are not safe because the value of pointers or reference may change by sending goroutine or by the receiving goroutine at the same time and the result is unpredicted.

SEND OPERATION

□ When you use pointers or references in the channel you must make sure that they can only access by the one goroutine at a time.

- **Mychannel <- element**
- **The above statement indicates that the data(element) send to the channel(Mychannel) with the help of a <- operator.**
- **The arrow points towards Mychannel and hence we are writing the data to channel Mychannel.**

RECEIVE OPERATION

- The receive operation is used to receive the data sent by the send operator.

  element := <-Mychannel

  - **The above statement indicates that the element receives data from the channel(Mychannel).**
  - **The arrow points outwards from Mychannel and hence we are reading from channel Mychannel and storing the value to the variable element.**

- You can also write a receive statement as:

  <-Mychannel

CLOSING A CHANNEL

 A **Channel** can be closed with the help of **close()** function. This is an in-built function and sets a flag which indicates that no more value will send to this channel.

 Syntax:

close()

CLOSING A CHANNEL

☐ We can also close the channel using for range loop. Here, the receiver goroutine can check the channel is open or close with the help of the given syntax:

ele, ok:= <- Mychannel

☐ If the value of ok is true which means the channel is open so, read operations can be performed. And if the value of is false which means the channel is closed so, read operations are not going to perform.

GO

IMPORTANT POINTS

- **Blocking Send and Receive:** In the channel when the data sent to a channel the control is blocked in that send statement until other goroutine reads from that channel. Similarly, when a channel receives data from the goroutine the read statement block until another goroutine statement.
- **Zero Value Channel:** The zero value of the channel is nil.
- **For loop in Channel:** A for loop can iterate over the sequential values sent on the channel until it closed.
- **Syntax:**

```
for item := range Chnl {
    // statements..
}
```

IMPORTANT POINTS

- **Length of the Channel:** In channel, you can find the length of the channel using **len()** function. Here, the length indicates the number of value queued in the channel buffer.
- **Capacity of the Channel:** In channel, you can find the capacity of the channel using **cap()** function. Here, the capacity indicates the size of the buffer.
- **Select and case statement in Channel:** In go language, select statement is just like a switch statement without any input parameter. This select statement is used in the channel to perform a single operation out of multiple operations provided by the case block.

UNIDIRECTIONAL CHANNEL

- A channel is a medium of communication between concurrently running goroutines so that they can send and receive data to each other.
- By default a channel is bidirectional but you can create a unidirectional channel also.
- A channel that can only receive data or a channel that can only send data is the unidirectional channel.

GO

UNIDIRECTIONAL CHANNEL

☐ The unidirectional channel can also create with the help of make() function as shown below:

// Only to send data
c1:= make(<- chan bool)

// Only to receive data
c2:= make(chan<-bool

GO

USE OF UNIDIRECTIONAL CHANNEL

- The unidirectional channel is used to provide the type-safety of the program so, that the program gives less error.

- Or you can also use a unidirectional channel when you want to create a channel that can only send or receive data.

GO

Session-06
Concurrency Patterns

DAY-3

CORE PACKAGES
BUILD REST API's
TESTING

Session-08
Building REST API's