MODULE-1

INTRODUCTION TO GO PROGRAMMING

1.Define Go programming language.

- Go is a general-purpose programming language with advanced features and a clean syntax.
- Because of its wide availability on a variety of platforms, its robust welldocumented common library, and its focus on good software engineering principles, Go is an ideal language to learn as your first programming language.
- Go is a cross-platform, open source programming language
- Go can be used to create high-performance applications
- Go is a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language
- Go was developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson in 2007
- Go's syntax is similar to C++

2. Explain Go Tools and write a simple program to print hello world using lang.

- Go is a compiled programming language, which means source code is translated into a language that your computer can understand. Therefore, before we can write a Go program, we need the Go compiler. The installer will setup Go for you automatically. We will be using version 1 of the language. Let's make sure everything is working. Open up a terminal and type the following

go version.

- ***go version go1.0.2***
- Your version number may be slightly different. If you get an error about the command not being recognized try restarting your computer. The Go tool suite is made up of several different commands and sub-commands. A list of those commands is available by typing:
- ***go help***

```
package main
import ("fmt")
func main() {
  fmt.Println("Hello World!")
}
```

OUTPUT: Hello World!

3. Explain two types of go program.

4. Explain Package Main, import fmt, and fmt.Println with example.

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

 fmt.Println("Hello World!")
}
```

OUTPUT: Hello World!

- **package main**
  • This is known as a “package declaration”. Every Go program must start with a package declaration. Packages are Go's way of organizing and reusing code.
  • There are two types of Go programs: executables and libraries.
  • Executable applications are the kinds of programs that we can run directly from the terminal. (in Windows they end with .exe)
  • Libraries are collections of code that we package together so that we can use them in other programs.
  • The next line is a blank line.

- **import ("fmt")**
  • The import keyword is how we include code from other packages to use with our program.
  • The fmt package (shorthand for format) implements formatting for input and output.
  • The import keyword is how we include code from other packages to use with our program. The fmt package (shorthand for format) implements formatting for input and output.
  • Comments are ignored by the Go compiler and are there for your own sake

- **func main() {**
  fmt.Println("Hello World!")
  }
  • Functions are the building blocks of a Go program.
  • They have inputs, outputs and a series of steps called statements which are executed in order.
  • This function has no parameters, doesn't return anything and has only one statement.
  • The name main is special because it's the function that gets called when you execute the program

5. Explain This statement components.

- First, we access another function inside of the fmt package called Println.
- Then we create a new string that contains Hello World and invoke (also known as call or execute) that function with the string as the first and only argument. At this point we've already seen a lot of new terminology and you may be a bit overwhelmed
- Sometimes it’s helpful to deliberately read your program out loud. One reading of the program we just wrote might go like this: Create a new executable program, which references the fmt library and contains one function called main

- That function takes no arguments, doesn't return anything and does the following: Access the Println function contained inside of the fmt package and invoke it using one argument – the string Hello World.

6. what is a comment? what are the two types of writing a comment.

- A comment is a text that is ignored upon execution.
- Comments can be used to explain the code, and to make it more readable.
- Comments can also be used to prevent code execution when testing an alternative code.

two types of writing a comment.

- **Single-line comment**
  Single-line comments start with two forward slashes (//).
  Any text between // and the end of the line is ignored by the compiler (will not be executed).

Example:

```
package main
import ("fmt")
func main() {
 fmt.Println("Hello World!") // This is a comment
}
```

- **Multi-line comment**
  Multi-line comments start with /* and ends with */.
  Any text between /* and */ will be ignored by the compiler:

Example:

```
package main
import ("fmt")
func main() {
 /* The code below will print Hello World
  to the screen, and it is amazing */
 fmt.Println("Hello World!")
}
```

7. Explain Numbers & Integers in go lang with example.

Integers

Integer data types are used to store a whole number without decimals, like 35, -50, or 1345000.

The integer data type has two categories:
Signed integers - can store both positive and negative values
Unsigned integers - can only store non-negative values

- **Signed integers**
  Signed integers, declared with one of the int keywords, can store both positive and negative values:
  Example:

```
package main
import ("fmt")

func main() {
  var x int = 500
  var y int = -4500
  fmt.Printf("Type: %T, value: %v", x, x)
  fmt.Printf("Type: %T, value: %v", y, y)
}
```

  OUTPUT: Type: int, value: 500
  Type: int, value: -4500

Go has five keywords/types of signed integers: int8, int16, int32 and int64.

- **Unsigned integers**
  Unsigned integers, declared with one of the uint keywords, can only store non-negative values:
  Example:

```
package main
import ("fmt")
func main() {
  var x uint = 500
  var y uint = 4500
  fmt.Printf("Type: %T, value: %v", x, x)
  fmt.Printf("Type: %T, value: %v", y, y)
}
```

  OUTPUT: Type: uint, value: 500
  Type: uint, value: 4500

Go has five keywords/types of signed integers: uint8, uint16, uint32 and uint64.

8. Explain floating point Numbers, integers with example

Floating point Numbers

Floating point numbers are numbers that contain a decimal component (real numbers).

- Floating point numbers are inexact. Occasionally it is not possible to represent a number.

- Like integers floating point numbers have a certain size (32 bit or 64 bit). Using a larger sized floating point number increases it's precision.
- In addition to numbers there are several other values which can be represented: “not a number” (NaN, for things like 0/0) and positive and negative infinity. (+∞ and −∞)
- Go has two floating point types: float32 and float64

Example:

```
package main
import ("fmt")
func main() {
 fmt.Println("1+1=”1+1)
}
```

OUTPUT:1+1-2

9. Explain strings with example.

The string data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example:

```
package main
import ("fmt")
func main() {
 fmt.Println (len("Hello World”))

 fmt.Println(“Hello World”)

 fmt.Println(“Hello”+”World”)
}
```

OUTPUT: 11

HelloWorld

Hello World

- A space is also considered a character, so the string's length is 11 not 10 and the 3rd line has "Hello " instead of "Hello".
- Concatenation uses the same symbol as addition. The Go compiler figures out what to do based on the types of the arguments. Since both sides of the + are strings the compiler assumes you mean concatenation and not addition.

10.Explain Boolean with example.

A boolean data type is declared with the bool keyword and can only take the values true or false.

The default value of a boolean data type is false.

Three logical operators are used with Boolean values: && and, || or, ! not.

Example:

```
package main
import ("fmt")
func main() {
 fmt.Println (true&&true)

 fmt.Println(true&&false)

 fmt.Println(true||true)

 fmt.Println(true||false)

 fmt.Println(!true)

}
```

OUTPUT: true false true true false

11.Explain variables in -lang with example.

- A variable is a storage location, with a specific type and an associated name.
- Variables in Go are created by first using the var keyword, then specifying the variable name (x), the type (string) and finally assigning a value to the variable (Hello World).

```
package main
import ("fmt")
func main() {
 var x string
 x=”Hello World”
 fmt.Println("x")
}
```

OUTPUT: Hello World.

- Variables in Go are similar to variables in algebra but there are some subtle differences:
- First when we see the = symbol we have a tendency to read that as “x equals the string Hello World”. There's nothing wrong with reading our program that way, but it's better to read it as “x takes the string Hello World” or “x is assigned the string Hello World”
-

```
package main
import ("fmt")
func main() {
 var x string
 x=”first”
 fmt.Println("x")
```

```
 x=”second”
 fmt.Println("x")
}
```

12.Explain different variables in go-lang with example.

- int- stores integers (whole numbers), such as 123 or -123
- float32- stores floating point numbers, with decimals, such as 19.99 or -19.99
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool- stores values with two states: true or false

Example:

```
package main
import ("fmt")
func main() {
 var a string
 var b int
 var c bool

 fmt.Println(a)
 fmt.Println(b)
 fmt.Println(c)
}
```

OUTPUT:0

False

13.Explain how to name a variable with example.

Go variable naming rules:

- A variable name must start with a letter or an underscore character (_)
- A variable name cannot start with a digit
- A variable name can only contain alpha-numeric characters and underscores (a-z, A-Z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- There is no limit on the length of the variable name
- A variable name cannot contain spaces
- The variable name cannot be any Go keywords

Example:

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

```
  x=”Puppy”
 fmt.Println("My Dog name is",x)
}
```

OUTPUT: My Dog name is:Puppy.

14.Explain scope in go lang.

The Scope of a variable can be defined as a part of the program where a particular variable is accessible. A variable can be defined in a class, method, loop, etc. Like C/C++, in Golang all identifiers are lexically (or statically) scoped, i.e. scope of a variable can be determined at compile time.

There are two types of scope:

- Local scope: The variable will be inside the function.
  Example:
  ```
  package main
  import “fmt”
  func main(){
    var x string=”Hello World”
     fmt.println(x)
  }
  ```
  OUTPUT: Hello World
- Global scope: The variable will be outside the function.
  Example:
  ```
  package main
  import “fmt”
  var x string=”Hello World”
  func main(){
     fmt.println(x)
  }
  ```
  OUTPUT: Hello World

15.Explain constant & different multiple variables in variable go lang

Constant:

If a variable should have a fixed value that cannot be changed, you can use the const keyword.

The const keyword declares the variable as "constant", which means that it is **unchangeable and read-only**.

Example:

```
package main
import “fmt”
func main(){
  const x string=”Hello World”
```

```
    fmt.println(x)
}
OUTPUT: Hello World
```

Constant Rules

- Constant names follow the same naming rules as variables
- Constant names are usually written in uppercase letters (for easy identification and differentiation from variables)
- Constants can be declared both inside and outside of a function

There are two types of constants:

- Typed constants
- Untyped constants

Multiple variables

In Go, it is possible to declare multiple variables in the same line.

Example:

```
package main
import ("fmt")

func main() {
  var a, b, c, d int = 1, 3, 5, 7

  fmt.Println(a)
  fmt.Println(b)
  fmt.Println(c)
  fmt.Println(d)
}
```

OUTPUT:

1 3 5 7

MODULE -2

1. Explain control structure in Arrays with example.

Now that we know how to use variables it's time to start writing some useful programs. First let's write a program that counts to 10, starting from 1, with each number on its own line. Using what we've learned so far, we could write this:

Example:

```
 package main
 import "fmt"
func main() {
  fmt.Println(1)
  fmt.Println(2)
  fmt.Println(3)
  fmt.Println(4)
  fmt.Println(5)
  fmt.Println(6)
  fmt.Println(7)
  fmt.Println(8)
  fmt.Println(9)
  fmt.Println(10)
}
```

OUTPUT: **1 2 3 4 5 6 7 8 9 10**

FOR: The for statement allows us to repeat a list of statements (a block) multiple times.

Example:

```
 package main
import "fmt"
func main() {
  i:=1
  for i<=10{
    fmt.Println(i)
    i=i+1
```

```
  }
}
```

OUTPUT: 1 2 3 4 5 6 7 8 9 10

• First, we create a variable called i that we use to store the number we want to print.

• Then we create a for loop by using the keyword for, providing a conditional expression which is either true or false and finally supplying a block to execute.

IF: • First, we need a way of determining whether or not a number is even or odd.

• An easy way to tell is to divide the number by 2.

• If you have nothing left over then the number is even, otherwise it's odd.

• So how do we find the remainder after division in Go?

• We use the % operator. 1 % 2 equals 1, 2 % 2 equals 0, 3 % 2 equals 1 and so on .

• Next, we need a way of choosing to do different things based on a condition

EXAMPLE:

```
package main
import "fmt"
func main() {
 for i:=1;i<=10;i++ {
   if i%2==0{
     fmt.Println(i,"even")
     }else{
      fmt.Println(i,"odd")
   }
 }
}
```

OUTPUT: 1 odd 2 even 3 odd 4 even 5 odd 6 even 7 odd 8 even 9 odd 10 even

2. Explain for statement in go long explain with example.

FOR: The for statement allows us to repeat a list of statements (a block) multiple times.

Example:

```
 package main
import "fmt"
func main() {
  i:=1
  for i<=10{
    fmt.Println(i)
    i=i+1
  }
}
```

OUTPUT: 1 2 3 4 5 6 7 8 9 10

• First, we create a variable called i that we use to store the number we want to print.

• Then we create a for loop by using the keyword for, providing a conditional expression which is either true or false and finally supplying a block to execute.

As an exercise lets walk through the program like a computer would:

• Create a variable named i with the value 1

• Is i <= 10? Yes.

• Print i

• Set i to i + 1 (i now equals 2)

• Is i <= 10? Yes.

• Print i

• Set i to i + 1 (i now equals 3)

• Set i to i + 1 (i now equals 11)

• Is i <= 10? No.

• Nothing left to do, so exit

3. Explain If in go lang Explain with example.

IF: • First, we need a way of determining whether or not a number is even or odd.

• An easy way to tell is to divide the number by 2.

• If you have nothing left over then the number is even, otherwise it's odd.

• So how do we find the remainder after division in Go?

• We use the % operator. 1 % 2 equals 1, 2 % 2 equals 0, 3 % 2 equals 1 and so on .

• Next, we need a way of choosing to do different things based on a condition

EXAMPLE:

```
package main

import "fmt"

func main() {
  for i:=1;i<=10;i++ {
    if i%2==0{
      fmt.Println(i,"even")
      }else{
       fmt.Println(i,"odd")
    }
  }
}
```

OUTPUT: 1 odd 2 even 3 odd 4 even 5 odd 6 even 7 odd 8 even 9 odd 10 even

Create a variable i of type int and give it the value 1

• Is i less than or equal to 10? Yes: jump to the block

• Is the remainder of i ÷ 2 equal to 0? No: jump to the else block

• Print i followed by odd

• Increment i (the statement after the condition)

• Is i less than or equal to 10? Yes: jump to the block

• Is the remainder of i ÷ 2 equal to 0? Yes: jump to the if block

• Print i followed by even.

4. Explains switch in go lang with syntax and with example.

The switch statement in Go is similar to the ones in C, C++, Java, JavaScript, and PHP. The difference is that it only runs the matched case so it does not need a break statement.

SYNTAX:

```
switch expression {
case x:
  // code block
case y:
  // code block
case z:
...
default:
  // code block
}
```

Example:

```
package main

import ("fmt")

func main() {

 day := 4

 switch day {

 case 1:

  fmt.Println("Monday")

 case 2:

  fmt.Println("Tuesday")

 case 3:

  fmt.Println("Wednesday")

 case 4:

  fmt.Println("Thursday")

 case 5:

  fmt.Println("Friday")

 case 6:

  fmt.Println("Saturday")

 case 7:

  fmt.Println("Sunday")

 }

}
```

OUTPUT: Thursday

5. write a program that prints out all the numbers clear divisible by 3 between 1& 100.

6. Explain Arrays, slices & Maps In go lang with example.

Arrays

Arrays are used to store multiple values of the same type in a single variable, instead of declaring separate variables for each value.

Declare an Array

In Go, there are two ways to declare an array:

1. With the var keyword:

SYNTAX:

var array_name = [length]datatype{values} // here length is defined

or

var array_name = [...]datatype{values} // here length is inferred

2. With the := sign:

array_name := [length]datatype{values} // here length is defined

or

array_name := [...]datatype{values} // here length is inferred

EXAMPLE:

```
package main
import ("fmt")
func main() {
  var arr1 = [3]int{1,2,3}
  arr2 := [5]int{4,5,6,7,8}
  fmt.Println(arr1)
  fmt.Println(arr2)
}
```

Output: [1 2 3]

[4 5 6 7 8]

Slices:

Slices are similar to arrays, but are more powerful and flexible.

Like arrays, slices are also used to store multiple values of the same type in a single variable.

However, unlike arrays, the length of a slice can grow and shrink as you see fit.

In Go, there are several ways to create a slice:

- Using the []*datatype*{*values*} format
- Create a slice from an array
- Using the make() function

SYNTAX:

```
myslice := []int{}
```

The code above declares an empty slice of 0 length and 0 capacity.

```
myslice := []int{1,2,3}
```

The code above declares a slice of integers of length 3 and also the capacity of 3.

In Go, there are two functions that can be used to return the length and capacity of a slice:

len() function - returns the length of the slice

cap() function - returns the capacity of the slice

EXAMPLE:

```
package main

import ("fmt")

func main() {

  myslice1 := []int{}

  fmt.Println(len(myslice1))

  fmt.Println(cap(myslice1))

  fmt.Println(myslice1)

  myslice2 := []string{"Go", "Slices", "Are", "Powerful"}

  fmt.Println(len(myslice2))

  fmt.Println(cap(myslice2))

  fmt.Println(myslice2)  }
```

OUTPUT:

0

0

[]

4

4

[Go Slices Are Powerful]

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.

EXAMPLE:

```
package main

import "fmt"

func main() {

  elements:=make(map[string]string)

  elements["H"]="Hydrogen"

  elements["He"]="Helium"

  elements["Li"]="Lithium"

  elements["B"]="Boran"

  elements["C"]="Carbon"

  elements["O"]="Oxygen"

  fmt.Println(elements["Li"])

}
```

OUTPUT: Lithium.

- elements is a map that represents the first 10 chemical elements indexed by their symbol. This is a very common way of using maps: as a lookup table or a dictionary.

- If you run this you should see nothing returned. Technically a map returns the zero value for the value type (which for strings is the empty string). Although we could check for the zero value in a condition (elements["Un"] == "") Go provides a better way
- Accessing an element of a map can return two values instead of just one. The first value is the result of the lookup, the second tells us whether or not the lookup was successful. In Go we often see code like this:
- First, we try to get the value from the map, then if it's successful we run the code inside of the block.

7. Write a program that uses arrays.

```
package main

import "fmt"

func main() {

  var x[5]float64

  x[0]=98

  x[1]=93

  x[2]=77

  x[3]=82

  x[4]=83

  var total float64=0

  for i:=0;i<5;i++ {

    total +=x[i]

   }

   fmt.Println(total/5)

}
```

OUTPUT:86.6

8. Explain slices in go lang with example.

Answer in 6Th question(Repeted)

9. Explain slices function in go with example.

Go includes two built-in functions to assist with slices: append and copy.

Example:

```
package main

import "fmt"

func main() {

 slice1:=[]int{1,2,3}

 slice2:=append(slice1,4,5)

 fmt.Println(slice1,slice2)

}
```

OUTPUT: [1 2 3] [1 2 3 4 5]

• After running this program slice1 has [1,2,3] and slice2 has [1,2,3,4,5]. append creates a new slice by taking an existing slice (the first argument) and appending all the following arguments to it.

• **Here is an example of copy:**

```
 package main

import "fmt"

func main() {

 slice1:=[]int{1,2,3}

 slice2:=append(slice1,4,5)

 copy(slice2,slice1)

 fmt.Println(slice1,slice2)

}
```

OUTPUT: [1 2 3] [1 2 3 4 5]

• After running this program slice1 has [1,2,3] and slice2 has [1,2]. The contents of slice1 are copied into slice2, but since slice2 has room for only two elements only the first two elements of slice1 are copied.

10.Explain Maps in go with example.

Answer in 6Th question(Repeted)

11.How do you access the 4th element of element of an array or slice.

12.Explain function with example.

- A function is a block of statements that can be used repeatedly in a program.
- A function will not execute automatically when a page loads.
- A function will be executed by a call to the function.

CREATE FUNCTION

- Use the func keyword.
- Specify a name for the function, followed by parentheses ().
- Finally, add code that defines what the function should do, inside curly braces {}.

SYNTAX

```
func FunctionName() {
// code to be executed
}
```

EXAMPLE:

```
package main

import ("fmt")

func myMessage() {

  fmt.Println("I just got executed!")

}

func main() {

  myMessage() // call the function

}
```

OUTPUT: I just got executed.

Naming Rules for Go Functions

- A function name must start with a letter

- A function name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Function names are case-sensitive
- A function name cannot contain spaces
- If the function name consists of multiple words, techniques introduced for multi-word variable naming can be used

13.How to return multiple values in variable function with example

Go is also capable of returning multiple values from a function:

```
package main

import ("fmt")

func myFunction(x int, y string) (result int, txt1 string) {

 result = x + x

 txt1 = y + " World!"

 return

}

func main() {

 fmt.Println(myFunction(5, "Hello"))

}
```

OUTPUT: 10 Hello World!

14.Example closure & recursive with example.

Closure

It is possible to create functions inside of functions

```
package main

import "fmt"

func main(){

 x:=0
```

```
 increment:=func() int{
    x++
    return x
  }
  fmt.Println(increment())
  fmt.Println(increment())
}
```

OUTPUT:1 2

• increment adds 1 to the variable x which is defined in the main function's scope. This x variable can be accessed and modified by the increment function. This is why the first time we call increment we see 1 displayed, but the second time we call it we see 2 displayed.

• A function like this together with the non-local variables it references is known as a closure. In this case increment and the variable x form the closure.

Recursive

Go accepts recursion functions. A function is recursive if it calls itself and reaches a stop condition.

```
package main
import ("fmt")
func testcount(x int) int {
 if x == 11 {
   return 0
 }
 fmt.Println(x)
 return testcount(x + 1)
}
func main(){
 testcount(1)
}
```

OUTPUT: 1 2 3 4 5 6 7 8 9 10

15.Write a function with are variable paraments that finds the greater numbers in a list of numbers

MODULE -3

1.Explain structures in go lang with example

- A struct (short for structure) is used to create a collection of members of different data types, into a single variable.
- While arrays are used to store multiple values of the same data type into a single variable, structs are used to store multiple values of different data types into a single variable.
- A struct can be useful for grouping data together to create records.

Syntax:

```
type struct_name struct {
  member1 datatype;
  member2 datatype;
  member3 datatype;
  ...
}
```

Example:

```
type Person struct {
  name string
  age int
  job string
  salary int
}
```

2. Write a program to declaring a structure in go lang.

To declare a structure in Go, use the type and struct keywords:

Syntax:

```
type struct_name struct {
  member1 datatype;
  member2 datatype;
  member3 datatype;
  ...
}
```

Example:

```
type Person struct {
  name string
  age int
  job string
  salary int
}
```

3. How to define a structure explain with example.

EXAMPLE:

```
package main
import "fmt"
type Address struct{
  Name string
  City string
  Pincode int
}
func main(){
```

```
 var a Address
 fmt.Println(a)
 a1:=Address{"Akshay","Deharadun",3623572}
 fmt.Println("Address1:",a1)
 a2:=Address{Name:"Anikaaa",City:"Ballia",Pincode:277001}
 fmt.Println("Address2:",a2)
 a3:=Address{Name:"Delhi"}
 fmt.Println("Address3:",a3)
}
```

OUTPUT: { 0}

Address1: {Akshay Deharadun 3623572}

Address2: {Anikaaa Ballia 277001}

Address3: {Delhi 0}

4. How to access fields of a struct explain with example.

To access individual fields of a struct you have to use dot (.) operator.

Example:

```
package main
import "fmt"
type Car struct {
 Name,Model,color string
 Weightlnkg  float64
}
func main(){
 c:=Car{Name:"Ferrari",Model:"GTC4",color:"Red",Weightlnkg:1920}
 fmt.Println("Car Name:",c.Name)
```

```
 fmt.Println("Car Color:",c.color)

 c.color="Black"

 fmt.Println("Car:",c)

}
```

OUTPUT: Car Name: Ferrari

Car Color: Red

Car: {Ferrari GTC4 Black 1920}

5. Explains pointers to a struct with example.

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

You can also create a pointer to a struct as shown in the below example:

Example:

```
package main

import "fmt"

type Employee struct{

 firstName,lastName string

 age,salary int

}

func main(){

 emp8:=&Employee{"sam","Anderson",55,60000}

 fmt.Println("FirstName:",(*emp8).firstName)

 fmt.Println("Age:",(*emp8).age)

}
```

Output: **First Name: Sam Age: 55**

6. Explain Nested structure in go lang with Syntax & example.

- A structure or struct in Golang is a user-defined type, which allows us to create a group of elements of different types into a single unit.
- Any real-world entity which has some set of properties or fields can be represented as a struct.
- Go language allows nested structure.
- A structure which is the field of another structure is known as Nested Structure. Or in other words, a structure within another structure is known as a Nested Structure.

SYNTAX:

type struct_name_1 struct{

// Fields }

type struct_name_2 struct{

variable_name struct_name_1 }

EXAMPLE:

```
package main
import "fmt"
type Student struct {
 name string
 branch string
 year int
}
type Teacher struct {
 name string
 subject string
 exp int
 details Student
}
func main() {
 result := Teacher{
 name: "Suman",
 subject: "Java",
 exp: 5,
 details: Student{"Bongo", "CSE", 2},
 }
 fmt.Println("Details of the Teacher")
 fmt.Println("Teacher's name: ", result.name)
 fmt.Println("Subject: ", result.subject)
 fmt.Println("Experience: ", result.exp)
 fmt.Println("\nDetails of Student")
 fmt.Println("Student's name: ", result.details.name)
 fmt.Println("Student's branch name: ", result.details.branch)
 fmt.Println("Year: ", result.details.year)
}
```

Output:

Details of the Teacher

Teacher's name: Suman

Subject: Java

Experience: 5

Details of Student

Student's name: Bongo

Student's branch name: CSE

Year: 2

7. Explain Anonymous structure and field in go lang, Explain with the help of example

Anonymous structure

In Go language, you are allowed to create an anonymous structure. An anonymous structure is a structure which does not contain a name. It useful when you want to create a one-time usable structure

syntax:

variable_name := struct{

  // fields }

{// Field_values}

EXAMPLE:

```
package main
import "fmt"
func main() {
 Element := struct {
 name string
 branch string
 language string
 Particles int
 }{
 name: "Pikachu",
 branch: "ECE",
 language: "C++",
 Particles: 498,
 }
```

```
 fmt.Println(Element)
}
```

Output: **{Pikachu ECE C++ 498}**

Anonymous field

- In a Go structure, you are allowed to create anonymous fields.
- Anonymous fields are those fields which do not contain any name you just simply mention the type of the fields and Go will automatically use the type as the name of the field.

SYNTAX:

type struct_name struct{

int

bool

float64 }

EXAMPLE:

```
package main
import "fmt"
type student struct {
 int
 string
 float64
}
func main() {
 value := student{123, "Bud", 8900.23}
 fmt.Println("Enrollment number : ", value.int)
 fmt.Println("Student name : ", value.string)
 fmt.Println("Package price : ", value.float64)
}
```

Output: Enrollment number : 123

Student name : Bud

Package price : 8900.23

8. Explains Anonymous fields. explain with the help of example

Same 7th question answer

9. Explain goroutines with example.

- Go language provides a special feature known as a Goroutines.
- A Goroutine is a function or method which executes independently and simultaneously in connection with any other Goroutines present in your program. Or in other words, every concurrently executing activity in Go language is known as a Goroutines.
- You can consider a Goroutine like a light weighted thread. The cost of creating Goroutines is very small as compared to the thread.
- Every program contains at least a single Goroutine and that Goroutine is known as the main Goroutine.
- All the Goroutines are working under the main Goroutines if the main Goroutine terminated, then all the goroutine present in the program also terminated. Goroutine always works in the background

EXAMPLE:

```
package main
import "fmt"
func display(str string) {
 for w := 0; w < 6; w++ {
 fmt.Println(str)
 }
}
func main() {
 // Calling Goroutine
 go display("Welcome")
 // Calling normal function
 display("Bvoc")
}
```

OUTPUT: Bvoc Bvoc Bvoc Bvoc Bvoc Bvoc

10.How to create a goroutines explain with syntax & example.

You can create your own Goroutine simply by using go keyword as a prefixing to the function or method call as shown in the below syntax:

SYNTAX:

func name(){

// statements

}

// using go keyword as the

// prefix of your function call

go name()

EXAMPLE:

```go
package main
import "fmt"
func display(str string) {
 for w := 0; w < 6; w++ {
 fmt.Println(str)
 }
}
func main() {
 // Calling Goroutine
 go display("Welcome")
 // Calling normal function
 display("Bvoc")
}
```

OUTPUT: Bvoc Bvoc Bvoc Bvoc Bvoc Bvoc

In the above example,

- we simply create a display() function and then call this function in two different ways first one is a Goroutine, i.e. go display("Welcome") and another one is a normal function, i.e. display("Bvoc").
- But there is a problem, it only displays the result of the normal function that does not display the result of Goroutine because when a new Goroutine executed, the Goroutine call return immediately.
- The control does not wait for Goroutine to complete their execution just like normal function they always move forward to the next line after the Goroutine call and ignores the value returned by the Goroutine

11.List the advantage of goroutines.

- Goroutines are cheaper than threads.
- Goroutine are stored in the stack and the size of the stack can grow and shrink according to the requirement of the program. But in threads, the size of the stack is fixed.
- Goroutines can communicate using the channel and these channels are specially designed to prevent race conditions when accessing shared memory using Goroutines.
- Suppose a program has one thread, and that thread has many Goroutines associated with it. If any of Goroutine blocks the thread due to resource requirement then all the remaining Goroutines will assign to a newly created OS thread. All these details are hidden from the programmers.

12.Explain Anonymous goroutine with syntax and example.

In Go language, you can also start Goroutine for an anonymous function or in other words, you can create an anonymous Goroutine simply by using go keyword as a prefix of that function.

Syntax

// Anonymous function call

go func (parameter_list){

// statement

}(arguments)

EXAMPLE:

```
package main
import (
 "fmt"
 "time"
)
func main() {
 fmt.Println("Welcome!! to Main function")
 go func() {
 fmt.Println("Welcome!! to bvoc")
 }()
 time.Sleep(1 * time.Second)
 fmt.Println("GoodBye!! to Main function")
}
```

Output: Welcome!! to Main function

Welcome!! to bvoc

GoodBye!! to Main function

13.Explain go - range keyword and write a program to show how to use a range.

- The range keyword is used in for loop to iterate over items of an array, slice, channel or map.
- With array and slices, it returns the index of the item as integer. With maps, it returns the key of the next key-value pair.
- Range either returns one value or two. If only one value is used on the left of a range expression.

Program:

```
package main
import "fmt"
func main() {
 /* create a slice */
 numbers := []int{0,1,2,3,4,5,6,7,8}
 /* print the numbers */
 for i:= range numbers {
 fmt.Println("Slice item",i,"is",numbers[i])
 }
 /* create a map*/
```

```
 countryCapitalMap := map[string] string
{"France":"Paris","Italy":"Rome","Japan":"Tokyo"}
 /* print map using keys*/
 for country := range countryCapitalMap {
 fmt.Println("Capital of",country,"is",countryCapitalMap[country])
 }
 /* print map using key-value*/
 for country,capital := range countryCapitalMap {
 fmt.Println("Capital of",country,"is",capital)
 }
}
```

OUTPUT:

Slice item 0 is 0

Slice item 1 is 1

Slice item 2 is 2

Slice item 3 is 3

Slice item 4 is 4

Slice item 5 is 5

Slice item 6 is 6

Slice item 7 is 7

Slice item 8 is 8

Capital of France is Paris

Capital of Italy is Rome

Capital of Japan is Tokyo

Capital of France is Paris

Capital of Italy is Rome

Capital of Japan is Tokyo

14.Explain go-Maps and how to define a go Map with example.

- Go provides another important data type named map which maps unique keys to values.
- A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

define a go Map

- You must use make function to create a map.
- declare a variable, by default map will be nil*/
- var map_variable map[key_data_type]value_data_type
- /* define the map as nil map can not be assigned any value*/
- map_variable = make(map[key_data_type]value_data_type)

EXAMPLE:

```
package main
import "fmt"
func main() {
var countryCapitalMap map[string]string
/* create a map*/
countryCapitalMap = make(map[string]string)
/* insert key-value pairs in the map*/
countryCapitalMap["France"] = "Paris"
countryCapitalMap["Italy"] = "Rome"
countryCapitalMap["Japan"] = "Tokyo"
countryCapitalMap["India"] = "New Delhi"
/* print map using keys*/
for country := range countryCapitalMap {
fmt.Println("Capital of",country,"is",countryCapitalMap[country])
}
/* test if entry is present in the map or not*/
capital, ok := countryCapitalMap["United States"]
/* if ok is true, entry is present otherwise entry is absent*/
if(ok){
fmt.Println("Capital of United States is", capital)
} else {
fmt.Println("Capital of United States is not present")
}
}
```

OUTPUT:

Capital of Japan is Tokyo

Capital of India is New Delhi

Capital of France is Paris

Capital of Italy is Rome

Capital of United States is not present

15.Example Go - recursion with example.

Recursion is the process of repeating items in a self-similar way. The same concept applies in programming languages as well. If a program allows to call a function inside the same function, then it is called a recursive function call.

Take a look at the following example –

```
func recursion() {

    recursion() /* function calls itself */ }

func main() {

     recursion() }
```

- The Go programming language supports recursion.
- That is, it allows a function to call itself.
- But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go on to become an infinite loop

EXAMPLE:

```
package main
import "fmt"
func factorial(i int)int {
 if(i <= 1) {
 return 1
 }
 return i * factorial(i - 1)
}
func main() {
 var i int = 15
 fmt.Printf("Factorial of %d is %d", i, factorial(i))
}
```

OUTPUT: Factorial of 15 is 1307674368000

16.Write a program of Fibonacci series using recursion in go.

```
package main
import "fmt"
func fibonaci(i int) (ret int) {
 if i == 0 {
    return 0
}
 if i == 1 {
    return 1
 }
 return fibonaci(i-1) + fibonaci(i-2)
}
func main() {
    var i int
    for i = 0; i < 10; i++ {
        fmt.Printf("%d ", fibonaci(i))
    }
}
```

OUTPUT: 0 1 1 2 3 5 8 13 21 34

MODULE -4

1. Explain package in Go language.

Packages are the most powerful part of the Go language.

- The purpose of a package is to design and maintain a large number of programs by grouping related features together into single units so that they can be easy to maintain and understand and independent of the other package programs.
- This modularity allows them to share and reuse.
- In Go language, every package is defined with a different name and that name is close to their functionality like “strings” package and it contains methods and functions that only related to strings.
- Every Go source file belongs to a package. To declare a source file to be part of a package, we use the following syntax –

package <Package name>

Every Go program starts with the main package. Whenever the compiler sees the main package, it treats the program as the executable code.

2. Explain import path and Package Declaration in Go language.

import path

In Go language, every package is defined by a unique string and this string is known as import path. With the help of an import path, you can import packages in your program. For example:

Import “fmt”

This statement states that you are importing an fmt package in your program. The import path of packages is globally unique. To avoid conflict between the path of the packages other than the standard library, the package path should start with the internet domain name of the organization that owns or host the package.

For example: import "geeksforgeeks.com/example/strings"

Package Declaration

In Go language, package declaration is always present at the beginning of the source file and the purpose of this declaration is to determine the default identifier for that package when it is imported by another package.

For example: package main

3. Explain Import Declaration & Blank Import in Go language.

Import Declaration

The import declaration immediately comes after the package declaration. The Go source file contains zero or more import declaration and each import declaration specifies the path of one or more packages in the parentheses.

For example:

```
// Importing single package

import "fmt"

// Importing multiple packages

import(   "fmt"

          "strings"

         "bytes" )
```

When you import a package in your program you're allowed to access the members of that package. For example, we have a package named as a "sort", so when you import this package in your program you are allowed to access sort.Float64s(), sort.SearchStrings(), etc functions of that package.

Blank Import

In Go programming, sometimes we import some packages in our program, but we do not use them in our program. When you run such types of programs that contain unused packages, then the compiler will give an error. So, to avoid this error, we use a blank identifier before the name of the package.

For example: import _ "strings"

It is known as blank import. It is used in many or some occasions when the main program can enable the optional features provided by the blank importing additional packages at the compile time.

4. Explain Nested packages & Giving names to the packages.

Nested packages

In Go language, you are allowed to create a package inside another package simply by creating a subdirectory. And the nested package can import just like the root package.

For example: import "math/cmplx"

Here, the math package is the main package and cmplx package is the nested package.

Sometimes some packages may have the same names, but the path of such type of packages is always different. For example, both math and crypto packages contain a rand named package, but the path of this package is different, i.e, math/rand and crypto/rand.

In Go programming, why always the main package is present on the top of the program? Because the main package tells the go build that it must activate the linker to make an executable file.

Giving names to the packages

In Go language, when you name a package you must always follow the following points:

- When you create a package the name of the package must be short and simple. For example strings, time, flag, etc. are standard library package.
- The package name should be descriptive and unambiguous.
- Always try to avoid choosing names that are commonly used or used for local relative variables
- The name of the package generally in the singular form. Sometimes some packages named in plural form like strings, bytes, buffers, etc. Because to avoid conflicts with the keywords.
- Always avoid package names that already have other connotations.

5. Write a go program to illustrate the concept of packages in package declaration.

```
package main
import (
    "bytes"
    "fmt"
    "sort"
)
func main() {
    slice_1 := []byte{'*', 'G', 'o', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm',
'i', 'n', 'g', '^', '^'}
    slice_2 := []string{"Go", "Gopher", "format", "variables", "structs"}
    fmt.Println("Original Slice:")
    fmt.Printf("Slice 1 : %s", slice_1)
    fmt.Println("\nSlice 2: ", slice_2)
    res := bytes.Trim(slice_1, "*^")
    fmt.Printf("\nNew Slice : %s", res)
    sort.Strings(slice_2)
    fmt.Println("\nSorted slice:", slice_2)
}
```

OUTPUT:

Original Slice:

Slice 1 : *GoProgramming^^

Slice 2: [Go Gopher format variables structs]

New Slice : GoProgramming

Sorted slice: [Go Gopher format structs variables]

6. Explain the Reflection Package and explain reflect. Copy() with example

- Reflection in Go is a form of metaprogramming. Reflection allows us to examine types at runtime.
- It also provides the ability to examine, modify, and create variables, functions, and structs at runtime.
- The Go reflect pacakge gives you features to inspect and manipulate an object at runtime.
- Reflection is an extremely powerful tool for developers and extends the horizon of any programming language.
- Types, Kinds and Values are three important pieces of reflection that are used in order to find out information.

The functions of Reflection Package:

1. reflect.Copy()

2. reflect.DeepEqual()

3. reflect.Swapper()

4. reflect.TypeOf()

5. reflect.ValueOf()

6. reflect.NumField()

7. reflect.Field()

8. reflect.FieldByIndex()

9. reflect.FieldByName()

10. reflect.MakeSlice()

11. reflect.MakeMap()

12. reflect.MakeChan()

13. reflect.MakeFunc()

reflect. Copy()

- Copy copies the contents of source into destination until either destination has been filled or source has been exhausted.
- It returns the number of elements copied.
- Destination and source each must have kind Slice or Array, and destination and source must have the same element type.
  EXAMPLE:

```
package main
import (
 "fmt"
 "reflect"
)
func main() {
 destination := reflect.ValueOf([]string{"A", "B", "C"})
 source := reflect.ValueOf([]string{"G", "O", "P", "L"})
 // Copy() function is used and it returns the number of elements copied
 counter := reflect.Copy(destination, source)
 fmt.Println(counter)
 fmt.Println(source)
```

```
 fmt.Println(destination)
}
```

OUTPUT: 3

[G O P L]

[G O P]

7. List the different Golang tools play an essential role when it comes to design web application.

1. Go Vendor

2. Gonative

3. Depth

4. Checkstyle

5. Apicompat

6. Go-Swagger

7. Go-Callvis

8. Go Meta Liner

9. Go Simple

10. Grapes Tool

Depth : This tool is meant to facilitate the web developers in retrieving and visualizing the source code dependency trees in the go language. The depth tool can be used in your project as a particular package.

Checkstyle : This is another important tool of the go language. The go language checkstyle tool is inspired by the java checkstyle and the go language's golint tool. In the go language, this tool prints out the coding style suggestions. It also allows the go developer to check the file line function and file line-param number.

Apicompat : Apicompat is an important tool of the go language. This tool helps the developer in detecting the incompatible changes and backward in the go program. It also checks the exported declarations in the program. It avoids false positives.

Go-Swagger : Go swagger is designed from the impression of swagger 2.0, and the go swagger can serialize and DE-serialize the functions of swagger 2.0. Go swagger tool provides the complete suite of GoLang feature to the developer. The tool brings the API components to work with swagger API, i.e., server, client, and date model.

Gotest tool: The gotest is a tool that is used to generate the go tests in the program. It generates the go tests from the source code of your go program. It makes it easier for the developer to write the go tests.

8. Explain the test functions in Golang and code Coverage.

test functions: Test functions are essential but it not always obvious how we can run tests in a better and efficient way. Writing the same code over and over again can take a huge amount of time. This post aims to provide some insights into what can be done to reduce this problem.

code Coverage: Code coverage is the way to determine the test coverage of a package. Test coverage is a metric that provides an outline of how much of the functions are covered by tests.

The rules to follow :

To do test coverage in Go we create a test file by adding a _test suffix.

```
filename_test.go
```

Then we simply use test functions in that file. To generate the coverage report, the -cover flag is added after that.

```
go test -cover
```

This cover flag tells the test tool to generate coverage reports for the package we are testing.

9. Explain the profiling and Benchmark and explain the Benchmark rules in go

Profiling: Profiling is a form of analyzing the program for optimizable code or functions. In software engineering, it is an essential task since optimization is a key factor when developing an application. Avoiding memory leaks and optimizing for better performance is almost always a target for enterprise-level software.

Benchmark: A benchmark by the dictionary definition is a standard, with which the same type of things may be compared. In programming, benchmark tests the time of execution for an operation. For example, a complex function execution time or a simple function executed a million times can be considered for benchmarking.

Benchmark rules in Go :

To do benchmark the functions defined for the benchmark must be of the form as shown below.

```
func BenchmarkXXX(b *tesing.B){

    // do benchmark... }
```

When running the go test we must provide the -bench flag which will show the output. go test -bench=. // the dot is the regex matching everything