Module-3

Structures in Golang

- A structure or struct in Golang is a user-defined type that allows to group/combine items of possibly different types into a single type. Any real-world entity which has some set of properties/fields can be represented as a struct. This concept is generally compared with the classes in object-oriented programming. It can be termed as a lightweight class that does not support inheritance but supports composition.

- 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.

Declaring a structure:

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

- In the above, the ***type*** keyword introduces a new type.
- It is followed by the name of the type (*Address*) and the keyword *struct* to illustrate that we're defining a struct.
- The struct contains a list of various fields inside the curly braces. Each field has a name and a type.
- **Note:** We can also make them compact by combining the various fields of the same type as shown in the below example:

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

To Define a structure: The syntax for declaring a structure:
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}

Note:

- Always pass the field values in the same order in which they are declared in the struct. Also, you can’t initialize only a subset of fields with the above syntax.
- Go also supports the *name: value* syntax for initializing a struct (the order of fields is irrelevant when using this syntax). And this allows you to initialize only a subset of fields. All the uninitialized fields are set to their corresponding zero value.

Example:

var a = Address{Name:”Akshay”, street:”PremNagar”, state:”Uttarakhand”, Pincode:252636} //city:””

```
// Golang program to show how to

// declare and define the struct

package main

import "fmt"

// Defining a struct type

type Address struct {

    Name    string
    city    string

    Pincode int

}

func main() {

    // Declaring a variable of a `struct` type

    // All the struct fields are initialized

    // with their zero value

    var a Address

 fmt.Println(a)

    // Declaring and initializing a

    // struct using a struct literal

    a1 := Address{"Akshay", "Dehradun", 3623572}

    fmt.Println("Address1: ", a1)

    // Naming fields while

    // initializing a struct

    a2 := Address{Name: "Anikaa", city: "Ballia",

                              Pincode: 277001}

    fmt.Println("Address2: ", a2)

    // Uninitialized fields are set to
    // their corresponding zero-value

    a3 := Address{Name: "Delhi"}

    fmt.Println("Address3: ", a3)

}
```

Output:

```
{  0}
Address1:  {Akshay Dehradun 3623572}
Address2:  {Anikaa Ballia 277001}
Address3:  {Delhi  0}
```

How to access fields of a struct?

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

```
// Golang program to show how to

// access the fields of struct

package main

import "fmt"

// defining the struct

type Car struct {

    Name, Model, Color string

    WeightInKg         float64
}

// Main Function

func main() {

    c := Car{Name: "Ferrari", Model: "GTC4",

            Color: "Red", WeightInKg: 1920}

    // Accessing struct fields

    // using the dot operator

    fmt.Println("Car Name: ", c.Name)

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

    // Assigning a new value

    // to a struct field

    c.Color = "Black"

    // Displaying the result

    fmt.Println("Car: ", c)

}
```

Output:

```
Car Name:  Ferrari

Car Color:  Red

Car:  {Ferrari GTC4 Black 1920}
```

Pointers to a struct

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:

```
// Golang program to illustrate

// the pointer to struct

package main



import "fmt"



// defining a structure

type Employee struct {

    firstName, lastName string

    age, salary int

}

func main() {

    // passing the address of struct variable

    // emp8 is a pointer to the Employee struct

    emp8 := &Employee{"Sam", "Anderson", 55, 6000}

    // (*emp8).firstName is the syntax to access
    // the firstName field of the emp8 struct

    fmt.Println("First Name:", (*emp8).firstName)

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

}
```

Output:

```
First Name: Sam

Age: 55
```

The *Golang* gives us the option to use *emp8.firstName* instead of the explicit dereference *(*emp8).firstName* to access the *firstName* field. Example to show this is following:

```
// Golang program to illustrate

// the pointer to struct

package main

import "fmt"

// Defining a structure

type Employee struct {

    firstName, lastName string

    age, salary         int

}

// Main Function

func main() {
    // taking pointer to struct

    emp8 := &Employee{"Sam", "Anderson", 55, 6000}

    // emp8.firstName is used to access

    // the field firstName

    fmt.Println("First Name: ", emp8.firstName)

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

}
```

Output:

```
First Name:  Sam

Age:  55
```

Nested Structure in Golang

- 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


}
```

Let us discuss this concept with the help of the examples:

Output:

```
Details of Author
{{Sona ECE 2013}}
```

Example 2:

```
// Golang program to illustrate

// the nested structure

package main

import "fmt"

// Creating structure

type Student struct {

    name   string

    branch string

    year   int

}
```

```
// Creating nested structure
type Teacher struct {
    name    string
    subject string
    exp     int
    details Student
}
func main() {
    // Initializing the fields
    // of the structure
    result := Teacher{
        name:    "Suman",
        subject: "Java",
        exp:     5,
        details: Student{"Bongo", "CSE", 2},
    }
    // Display the values
    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
```

Anonymous Structure and Field in Golang

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.

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. You can create an anonymous structure using the following syntax:

```
variable_name := struct{

// fields

}{// Field_values}
```

Let us discuss this concept with the help of an example:

Example:

```
// Go program to illustrate the

// concept of anonymous structure

package main



import "fmt"



// Main function

func main() {



    // Creating and initializing

    // the anonymous structure

    Element := struct {
        name      string

        branch    string

        language  string

        Particles int

    }{

        name:      "Pikachu",

        branch:    "ECE",

        language:  "C++",

        Particles: 498,

    }



    // Display the anonymous structure

    fmt.Println(Element)

}
```

Output:

```
{Pikachu ECE C++ 498}
```

Anonymous Fields

- 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.
- You can create anonymous fields of the structure using the following syntax:

```
type struct_name struct{
   int
   bool
   float64
}
```

Important Points:

- In a structure, you are not allowed to create two or more fields of the same type as shown below:

```
type student struct{
int
int
}
```

If you try to do so, then the compiler will give an error.

- You are allowed to combine the anonymous fields with the named fields as shown below:

```
type student struct{
 name int
 price int
 string
}
```

Let us discuss the anonymous field concept with the help of an example:

Example:

```
// Go program to illustrate the

// concept of anonymous structure

package main

import "fmt"

// Creating a structure

// with anonymous fields

type student struct {

    int

    string

    float64

}

// Main function

func main() {

    // Assigning values to the anonymous

    // fields of the student structure

    value := student{123, "Bud", 8900.23}
    // Display the values of the fields

    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
```

Goroutines – Concurrency in Golang

- 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.

How to create a Goroutine?

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 program to illustrate

// the concept of Goroutine

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.
- So, to executes a Goroutine properly, we made some changes in our program as shown in the below code:

Modified Example:

```
// Go program to illustrate the concept of Goroutine

package main

import (

    "fmt"

    "time"

)

func display(str string) {

    for w := 0; w < 6; w++ {

        time.Sleep(1 * time.Second)

        fmt.Println(str)

    }

}

func main() {

    // Calling Goroutine

    go display("Welcome")

    // Calling normal function
    display("bvoc")

}
```

```
Output:
Welcome

bvoc

bvoc

Welcome

Welcome

bvoc

bvoc

Welcome

Welcome

bvoc

bvoc
```

- We added the *Sleep() method* in our program which makes the main Goroutine sleeps for 1 second in between 1-second the new Goroutine executes, displays "*welcome*" on the screen, and then terminate after 1-second main Goroutine re-schedule and perform its operation.
- This process continues until the value of the *z<6* after that the main Goroutine terminates. Here, both Goroutine and the normal function work concurrently.

Advantages 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.

Anonymous Goroutine

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 as shown in the below Syntax:

Syntax:

```
// Anonymous function call

go func (parameter_list){

// statement

}(arguments)
```

Example:

```
// Go program to illustrate how

// to create an anonymous Goroutine

package main

import (

    "fmt"

    "time"

)

// Main function

func main() {

    fmt.Println("Welcome!! to Main function")
    // Creating Anonymous Goroutine

    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
```

Go - 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, it is the 1st value in the following table.

Range expression	1st Value	2nd Value(Optional)
Array or slice a [n]E	index i int	a[i] E
String s string type	index i int	rune int

map m map[K]V	key k K	value m[k] V
channel c chan E	element e E	none

Example

The following paragraph shows how to use range −

```
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)
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
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
```

Go – Maps

- 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.

Defining a 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

The following example illustrates how to create and use a map −

```
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")
}
}
```

When the above code is compiled and executed, it produces the following result

```
Capital of India is New Delhi
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of United States is not present
```

- delete() Function
- delete() function is used to delete an entry from a map. It requires the map and the corresponding key which is to be deleted.

  For example −

```
package main

import "fmt"

func main() {
/* create a map*/
countryCapitalMap := map[string] string
{"France":"Paris","Italy":"Rome","Japan":"Tokyo","India":"New Delhi"}

fmt.Println("Original map")

/* print map */
for country := range countryCapitalMap {
fmt.Println("Capital of",country,"is",countryCapitalMap[country])
}

/* delete an entry */
delete(countryCapitalMap,"France");
fmt.Println("Entry for France is deleted")

fmt.Println("Updated map")

/* print map */
for country := range countryCapitalMap {
fmt.Println("Capital of",country,"is",countryCapitalMap[country])
```

```
   }
}
```

When the above code is compiled and executed, it produces the following result

```
Original Map
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Entry for France is deleted
Updated Map
Capital of India is New Delhi
Capital of Italy is Rome
Capital of Japan is Tokyo
```

Go - Recursion

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.

Examples of Recursion in Go

Recursive functions are very useful to solve many mathematical problems such as calculating factorial of a number, generating a Fibonacci series, etc.

Example 1: Calculating Factorial Using Recursion in Go

The following example calculates the factorial of a given number using a recursive function −

```
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))
}
```

When the above code is compiled and executed, it produces the following result −

```
Factorial of 15 is 1307674368000
```

Example 2: Fibonacci Series Using Recursion in Go

The following example shows how to generate a Fibonacci series of a given number using a recursive function −

```
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))
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
0 1 1 2 3 5 8 13 21 34
```