Unit-2 Variables

How to Name a Variable, Scope, Constants, Defining Multiple Variables, Control Structures - The for Statement, the if Statement, the switch Statement, Arrays, Slices, and Maps, Arrays, Slices - append, copy, Maps.

A variable is a storage containing data temporarily to work with it.

Naming Conventions for Golang Variables:

- A name must begin with a letter, and can have any number of additional letters and numbers.
- A variable name cannot start with a number or underscore.
- A variable name cannot contain spaces.
- Use camelcase letters ie; If a name consists of multiple words, each word after the first should be capitalized like this: empName, EmpAddress, etc.
- Variable names are case-sensitive (car, Car and CAR are three different variables).
- A variable name cannot be a reserved word as they are part of the Go syntax like int, type, for, etc.
- If the name of a variable begins with a lower-case letter, it can only be accessed within the current package this is considered as unexported variables.
- If the name of a variable begins with a capital letter, it can be accessed from packages outside the current package one this is considered as exported variables.
- Avoid using package names as variable names.

Syntax for Declaration of variable
The most general form to declare a variable in Golang uses the var keyword, an explicit type, and an assignment.

```
var variablename datatype = value
Variablename := value
```

Declaration with initialization

If you know what a variable's value will be, you can declare variables and assign them values on the same line.

```
package main
import "fmt"
func main() {
var num int = 10
var city string = "Hyderabad"
fmt.Println(num)
fmt.Println(city)
}
```

Declaration without initialization

```
package main
import "fmt"
func main() {
var num int
var city string

num = 10
city = "Hyderabad"

fmt.Println(num)
fmt.Println(city)
}
```

After the execution of the statements above, the variable num will hold the value 10 and thevariable city will hold the value Hyderabad.

Declaration with type inference

Variable can be declared without specifying type, when you are assigning a value to a variable. The type of the value assigned to the variable will be used as the type of thatvariable.

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

func main() {
var i = 10
var s = "Canada"

fmt.Println(reflect.TypeOf(i))
fmt.Println(reflect.TypeOf(s))
}
```

Short Variable Declaration in Golang

The := short variable assignment operator indicates that short variable declaration is beingused. There is no need to use the var keyword or declare the variable type.

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

func main() {
name := "John Doe"
fmt.Println(reflect.TypeOf(name))
}
```

Declaration of multiple variables

Multiple Variable Declarations Without Initialization

In Go, we can declare multiple variables of the same type simultaneously. In such cases, a default value (generally, it is 0) is assigned to all variables.

Syntax

```
var <variable_name1>, <variable_name2>    , <variable_nameN> <type>
```

```
package main
import "fmt"
func main() {
var val_1, val_2, val_3 int

fmt.Println("The value of val_1 is: ", val_1)
fmt.Println("The value of val_2 is: ", val_2)
fmt.Println("The value of val_3 is: ", val_3)

}
```

Multiple Variable Declarations With Initialization

We can declare multiple variables in Go and initialize some initial values for each.

Syntax

```
var <variable_name1>, <variable_name2>......, <variable_nameN> <type> = {value1}, {value2} ,
{valueN}
```

```
package main
import "fmt"
func main() {
        var val_1, val_2, val_3 int = 10, 11, 12

        fmt.Println("The value of val_1 is: ", val_1)
        fmt.Println("The value of val_2 is: ", val_2)
        fmt.Println("The value of val_3 is: ", val_3)

}
```

Declare Variables of Multiple Types

A single var keyword can be used to declare many variables of different data types.

```
Syntax var (
<variable_name1> <type> = {value1}
<variable_name2> <type> = {value2}
<variable_name3> <type> = {value3}

<variable_nameN> <type> = {valueN}
)
```

```
package main
 import "fmt"
func main() {
var (
val_1   int = 10
        message string
        val_2   int = 20
        )

        fmt.Println("The value of val_1 is: ", val_1)
        fmt.Println("The message is: ", message)
        fmt.Println("The value of val_2 is: ", val_2)

}
```

Scope of Golang Variables

In Go language, the scope of a variable refers to the region of the code where the variable isaccessible. In abstract terms, the scope of a variable is its lifetime in the program.

Golang variable scope can be divided into two categories based on where they were declared:

Local Variables
Global Variables

Local Variables

Golang Variables which are declared inside a function or a block of code are called Local Variables. These Variables are not visible outside the block of code.

These Variables are not accessible once the function or block of code is executed. There will be a compile-time error if Local variables are declared twice in the same scope.

```
package main
import "fmt"
 func main() {
// Local scope for the main method starts here.

// Declaring local variable

        var scalar string = "Welcome to scalar"

        // Printing local variable

        fmt.Printf(scalar)

    }
```

Global Variable

Global Variables are defined outside of a function or a block of code. Generally, they are declared at the top of our Go file. Global variables can be accessed from any function throughout the lifetime of the program and can be accessed by any function defined in our code.

```
package main
 import "fmt"
// Declaring Global variable

var scalar string = "Welcome to scalar" func main() {
        // Printing Global variable in main

        fmt.Println(scalar + " from main")

        //function call

        displayGreeting()

}

func displayGreeting() {

        // Printing Global variable from a random function

fmt.Println(scalar + " from function")

}
```

Constants in Golang

- Constants are the fixed values that cannot be changed once declared.
- A variable can vary during the runtime, whereas a constant does not change; it remains constant.
- This allows users to declare a variable that will remain constant during the program's compilation.

Constants are declared similarly to variables, but with the const keyword as a prefix to declare a constant of a specified type.

Syntax

```
const variable_name = value const variable_name type = value
package main import "fmt" func main() {
  const LENGTH int = 10
  const WIDTH int = 5
  var LENGTH int =20

  //error
  fmt.Printf("const variable value cannot be changed")
}
```

Types of Constants in Golang

The various types of constants in the Go language are as follows:

Typed and untyped numeric constant
Numeric Constant
String Literals
Boolean Constant

Typed and Untyped Numeric Constant

Go, constants can be specified with or without a type.

In simple words, Typed Numeric constants are defined by their data type. They can no longer hold values of any other data type once declared.

Untyped Numeric constants, on the other hand, are stated without specifying their data type. Type inference is used to determine the data type. The constant val1 is untyped, whereas val2 is typed into the program below.

```
package main
import "fmt"
func main() {
const val1 = 10
fmt.Println("The value of val1 is: ", val1) const val2 int = 20
fmt.Println("The value of val2 is: ", val2)
}
```

Numeric Constant

Numeric constants are values with a high degree of precision. Numeric Constant can also be classified into three types:

Integer-Integer constants are assigned integer values such as decimal, octal, and hexadecimal.
Floating point-Floating-point constants have integer, decimal, fractional, and exponent parts.
Complex- It is an ordered pair or real pair of integer constants (or parameters) separated by a comma, and the pair is enclosed in parentheses. The first constant represents the real part, while the second represents the imaginary part.

```
package main
 import "fmt"

func main() {

const val1 = 10 const val2 = 10.23
            const cmplx complex64 = 10

fmt.Printf("The val1 if of %T type \n", val1)

            fmt.Println("The value of val1 is: ", val1)
fmt.Printf("The val2 if of %T type \n", val2)
 fmt.Println("The value of val2 is: ", val2)
fmt.Printf("The complex number is: %v and is of type %T", cmplx, cmplx)
}
```

String Literals

Their values are wrapped in double quotations. Constants for typed strings contain the data type string in their syntax. Type inference determines the data type of untyped constants.

```
package mainimport "fmt" func main() {
const message1 = "Hello"

fmt.Printf("The message1 is: %s and is of type %T", message1, message1)

}
```

Boolean Constant

Boolean constants are similar to string constants. It follows the same principles as a string constant. The sole distinction is that it has two untyped constants, true and false.

```
package main
import "fmt"
func main() {
const myBool bool = true

fmt.Printf("The value of myBool is: %v", myBool)
fmt.Printf("\nmyBool is of type %T", myBool)
}
```

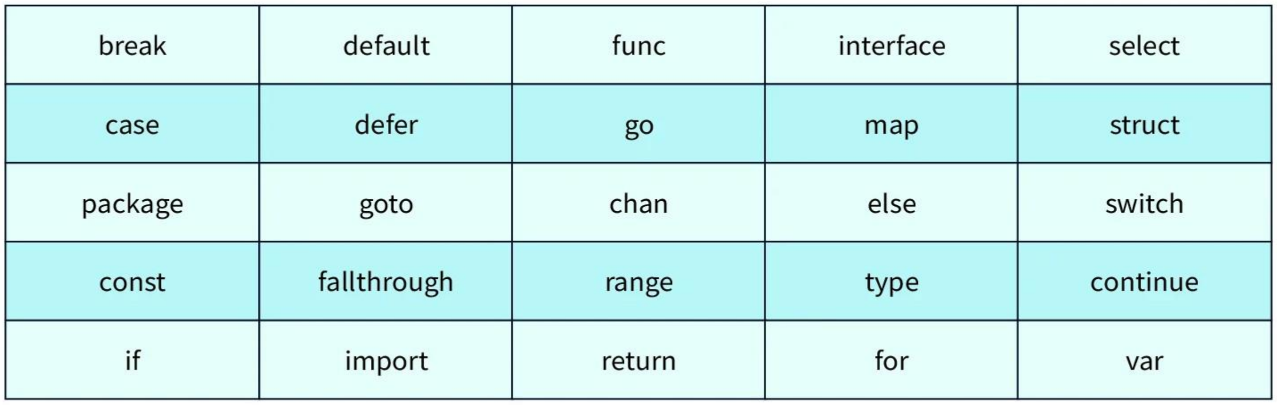
Keywords in Golang

Keywords, often known as reserved words, are terms in a programming language that are used for

internal processes or to indicate preset activities. As a result, certain words are not permitted to be used as identifiers. A compile-time error will be generated if you do this.

There are 25 keywords that are available in the go language. Inside the go program, each keyword has a specific task to do. The keyword cannot be used as a variable name or constant.

List of Golang Keywords



break	default	func	interface	select
case	defer	go	map	struct
package	goto	chan	else	switch
const	fallthrough	range	type	continue
if	import	return	for	var

Control Structures

Control statements in Golang are used to control the flow of the program's execution based on specified conditions.
These are used to make the execution flow advance and make the flow of code in a boolean expression.

For loop

For loop in Golang is used for repeating a block of code until a condition specified in the loop is met. For is the only loop available in Golang to execute a block of code repeatedly. In this, all three parameters are optional.

The flow control of the for loop is as follows:

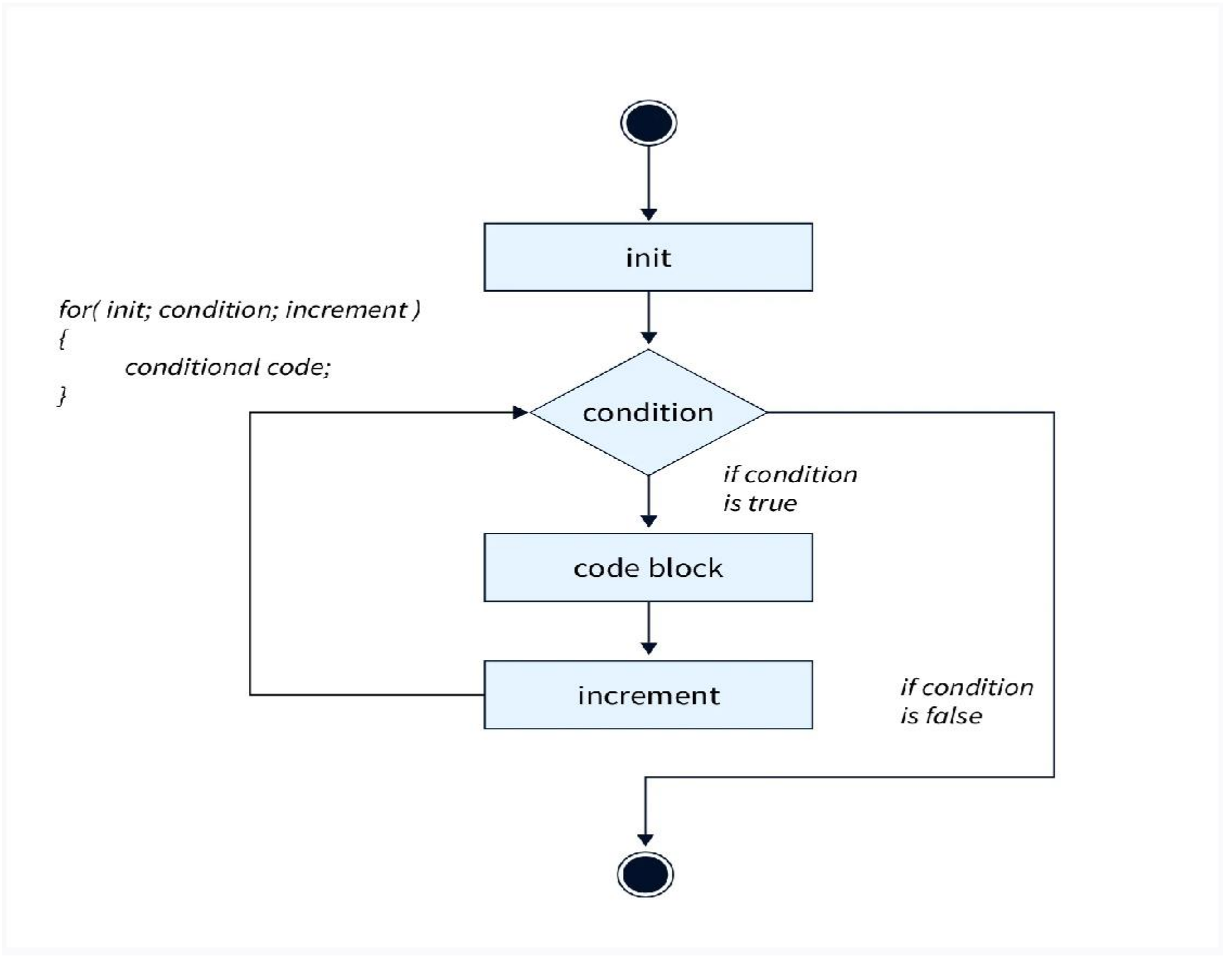
- If a condition is mentioned in the for loop then as long as the condition is true, the loop executes.
- A loop works on 3 parameters, initial state, condition, and a state change operation.

  - The first step is initializing the state, it executes only once. In this step, the initialization of any loop variable is done.
  - The next step is to evaluate the condition, the loop body is executed till the condition is true. If the condition i is false then the loop does not execute, instead, the control goes to the next statement after the for loop.
  - Once the body of the loop is executed or as a state change, the flow control goes back to the post-stated. The posted statement includes control variables that need to be updated.

The condition is however evaluated once again and if it's true, the loop is executed and the whole process is repeated. The loop terminates when the condition becomes false.

Syntax

```
for init; condition; post {
statement
}
These three are optional, if we skip init and post we can write: like while loop for condition {
        statement
}
If we skip the condition we can write: like an infinite loop for {
        statement
}
```



Program to print the first 10 natural numbers

Example 1

```
package main
 import "fmt"

func main() {

  // for loop terminates when the value of i becomes 11
  for i := 1; i <= 10; i++ {
            fmt.Println(i)
  }

}
```

Example 2

```
package main
import ("fmt")

func main() {
  n := 1

  for n <= 10 {
            fmt.Println(n)
            n++
  }
}
```

Example 3
Program to infinitely iterate through the loop

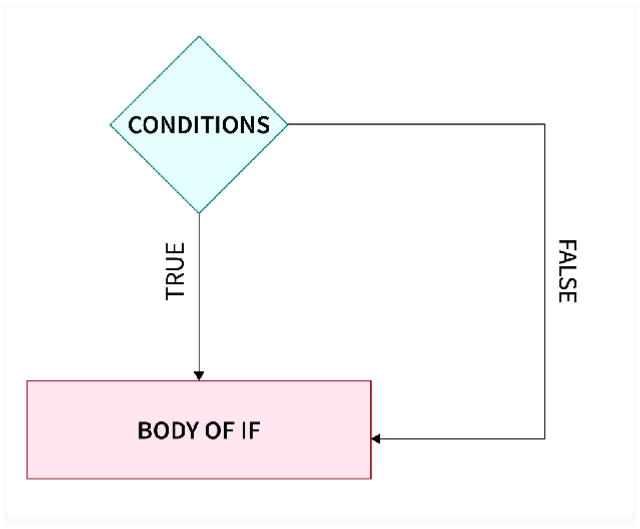
```
package main
 import "fmt"
func main() {
for {
fmt.println("Welcome to go programming")
}
}
```

If-statement

The if-statement is a fundamental control-flow statement in programming whose main aim is to make decisions.
The code block executes if the condition is true; else, it doesn't.
Syntax:

```
if condition {
	// do something...
}
```

Flow Chart:



Example

```
package main
 import (
	"fmt"
)

func main() {
	// taking a local variable
```

```
        var country = "India"
        x := true

        // checking the condition
        if x {
        fmt.Println(country)
        }
}
```

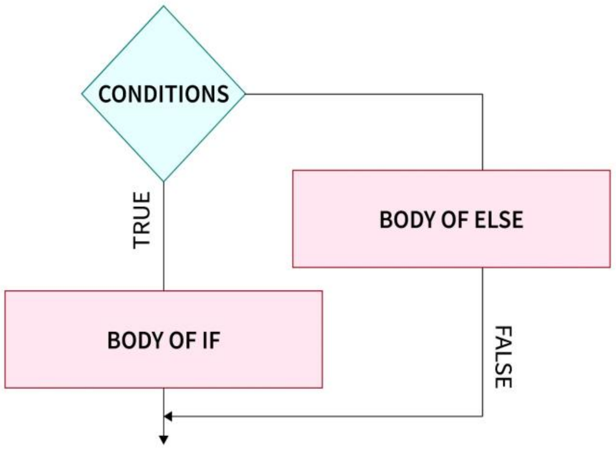
If..else Statement

The if else statement enables you to execute a block of code if the given condition is true and another block of code if it's false.

Syntax:

```
if  condition {

// code only executes if the condition is true

} else {

// code only executes if the condition is false

}
```

Flow Chart:



Example

```
package main
import (
        "fmt"
)

func main() {
   // taking a local variable
        x := 60
   // checking the condition
        if x == 100 {
        fmt.Println("Right Number")
        } else {
        fmt.Println("Wrong Number") //prints wrong number
        }
}
```

Nested..If

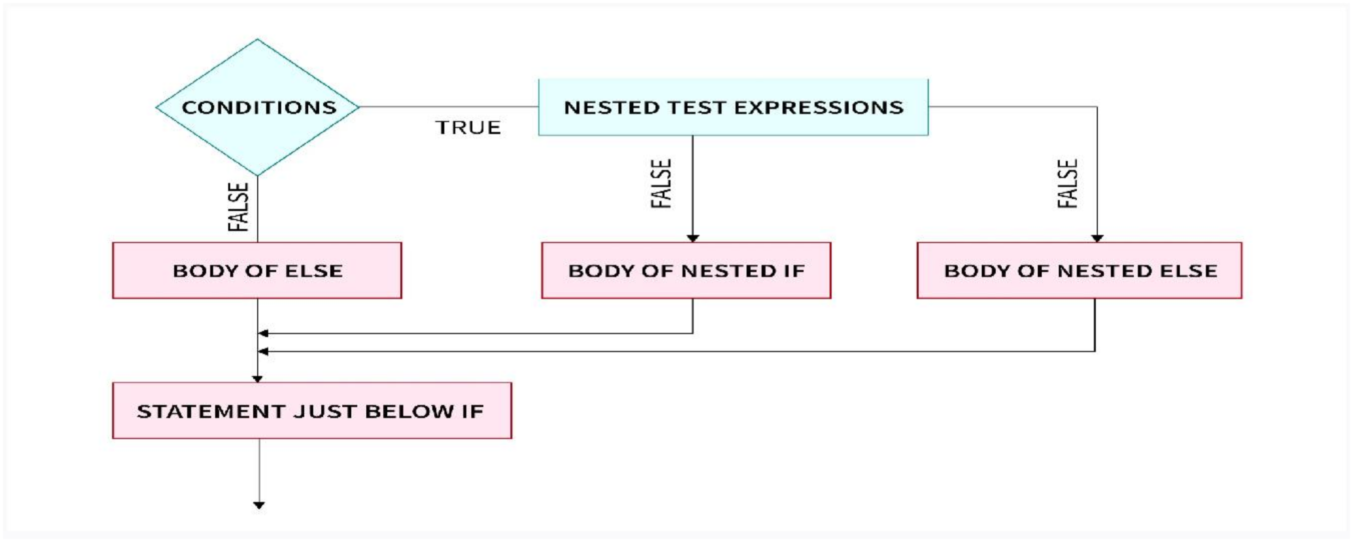
If statements can be freely nested to any depth.

An if statement can be contained by another if statement. In this section, we will look at an example of a nested if statement.

Syntax:

```
if condition_1 {

  // Executes when condition1 is true

  if condition_2 {

        // Executes when condition2 is true
  }
}
```

Flow Chart:



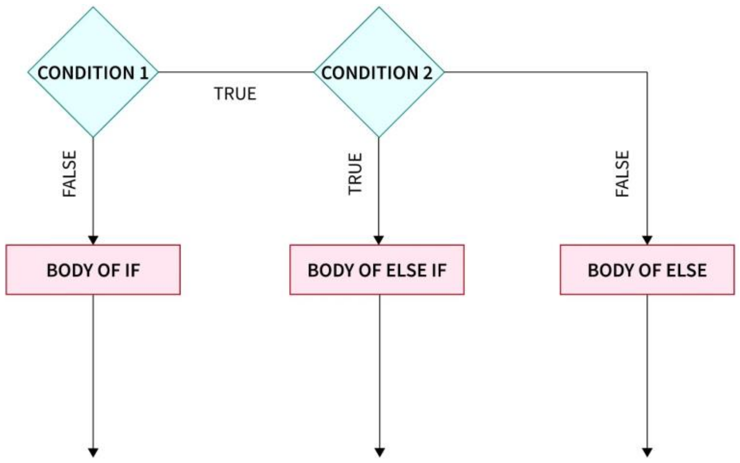
If..else..if Statement

example

```
package main
import (
        "fmt"
)
func main() {
   // taking a local variable
        var name string = "Scaler"
        var num int = 63
   // checking the condition
        if name == "Scaler" {
        if num == 63 {
        fmt.Println("Welcome to go programming.")
        } else {
        fmt.Println("Wrong text")
        }
        } else {
        fmt.Println("Who are you?")
        }

        // Welcome to Scaler
}
```

What ever condition in an if or else if returns true, the corresponding code block will be executed. Otherwise, if none of the conditions is met, the else block will execute.

Flow Chart:



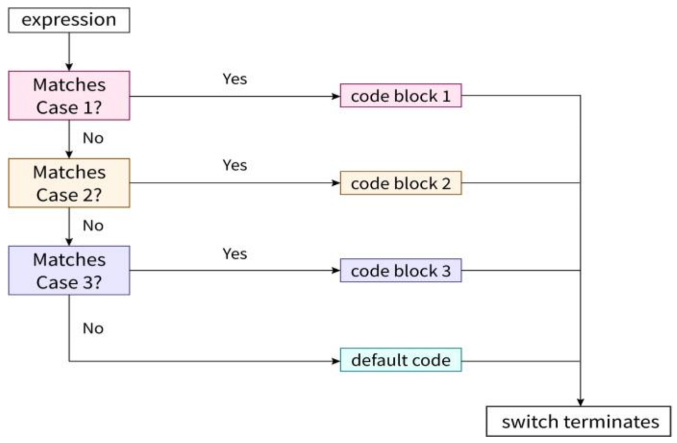
Syntax:

```
if condition_1 {
        // this block will execute
    // when condition_1 is true

        } else if condition_2 {

           // this block will execute
           // when condition2 is true
        }
        else {

                  // this block will execute when all the above condition is false
        }

        Example: package main
         import (
                  "fmt"
        )

        func main() {
                  num := 12
           // checking the condition

                  if num <= 81 {
                  // if a condition is true then
                  // print the following */
                  fmt.Println(num, "is less than or equal to 50")
                  } else if num >= 51 && num <= 100 {
                  fmt.Println(num, "is between 51 and 100")
                  // if none of the conditions is true print else block
                  } else {
                  fmt.Println(num, "is greater than 100")
                  }
        }
```

Switch Statement in Go

- A switch statement is generally a multi-way branch statement that transfers the execution to different parts of code depending on the value of the expression.
- Switch evaluates an expression and compares it with a list of possible matches and executes blocks of code according to the match.

flow chart:



when an expression is evaluated, it checks the first case, if it matches then the statement is executed else it moves on to case 2. Here it again checks if the case matches the expression, if it does then the statement is executed, else it goes on to the next statement and this process goes on for all the cases. If the expression doesn't match any of the cases then the default statement is executed.

Syntax:

```
switch expression {
case value_1:
statement case value_2:
statement
.
.
default: statement
}
```

Example

```
package main
import ("fmt")

func main() {
dayOfWeek:= 4

switch dayOfWeek {

case 1: fmt.Println("Sunday")

case 2: fmt.Println("Monday")

case 3: fmt.Println("Tuesday")

case 4: fmt.Println("Wednesday")

case 5: fmt.Println("Thursday")
case 6: fmt.Println("Friday")

case 7: fmt.Println("Saturday")
default: fmt.Println("Invalid day")
}
}
```

Using Multiple Cases

Inside a single case block, multiple statements can be used for evaluating the expression. The case block is executed if it matches any one value from the case value.

```
package main
import ("fmt")

func main() {
  dayOfWeek:= "Sunday"

  switch dayOfWeek {
        case "Saturday", "Sunday":
        fmt.Println("Weekend")

        case "Monday","Tuesday","Wednesday","Thursday","Friday":
        fmt.Println("Weekday")

        default:
        fmt.Println("Invalid day")
  }
}
```

Golang Arrays

- An array is a data structure that consists of a collection of elements of a single type,and whichcan hold more than one value at a time.
- The values an array holds are called its elements or items.
- An array holds a specific number of elements, and it cannot grow or shrink.
- The items in the array can be accessed through their index, It starts with zero.
- The number ofitems in the array is called the length or size of an array.

Different data types can be handled as elements in arrays such as Int, String, Boolean, andothers.

Syntax:

```
var  array_name  [len]datatype

var array_name = [size]datatype{element1 of array,element2 of array…..}
```

Example:

```
package main
import "fmt"
```

```
 func main() { var x [5]int
var i, j int
for i = 0; i < 5; i++ {x[i] = i + 10
fmt.Printf("Element[%d] = %d\n", i, x[i])

}
}
```

Initializing Arrays

Assign a single element of the array arrayname[index] = value myArray[2]=20
You can initialize array in Go either one by one or using a single statement as follows −

```
var number = [5]int32{1000, 200, 340, 70, 500}
```

Accessing Array Elements

An element is accessed by indexing the array name.
This is done by placing the index of the element within square brackets after the name ofthe array.
example

```
float32 salary = balance[9]
```

Go Multi Dimensional Arrays
Multi dimensional arrays is simply a list of one-dimensional arrays.

Sytax:

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type
```

Two-Dimensional Arrays

A two-dimensional array is the simplest form of a multidimensional array. Atwo-dimensional array is, in essence, a list of one-dimensional arrays.

```
var arrayName [ x ][ y ] variable_type
```

A two-dimensional array can be think as a table which will have x number of rows and ynumber of columns.
Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row.Following is an array with 3 rows and each row has 4 columns.

```
a = [3][4]int{
   {0, 1, 2, 3} ,/* initializers for row indexed by 0 */
   {4, 5, 6, 7} ,/* initializers for row indexed by 1 */
   {8, 9, 10, 11}         /* initializers for row indexed by 2 */
}
```

Accessing Two-Dimensional Array Elements

An element in two dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3]
```

The above statement will take the 4th element from the 3rd row of the array.

```
package main import "fmt" func main() {
   /* an array with 5 rows and 2 columns*/

   var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}

   var i, j int

   /* output each array element's value */

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

         for j = 0; j < 2; j++ {

         fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )

         }

   }

}
```

Go Slice

slice is a segmented view of an underlying array.

This segment can be the entire array or a subset of an array. We define the subset of anarray by indicating the start and end index.

Defining a slice

To define a slice, you can declare it as an array without specifying its size.

```
var numbers []int /* a slice of unspecified size */
```

Alternatively, you can use make function to create a slice.

```
/* numbers == []int{0,0,0,0,0}*/
numbers = make([]int,5,5) /* a slice of length 5 and capacity 5*/
```

```
package main
 import "fmt"
 func main() {
   slice := make([]int, 10)
   printSlice("slice", slice)
   slice1 := make([]int, 0, 10)
   printSlice("slice1", slice1)
   slice2 := slice1[:5]
   printSlice("slice2", slice2)
   slice3 := slice2[2:5]
   printSlice("slice3", slice3)
}
func printSlice(s string, x []int) {
fmt.Printf("%s length=%d capacity=%d %v\n", s, len(x), cap(x), x)
}
```

len() and cap() functions

The len() function returns the elements presents in the slice where cap() function returns the capacity of the slice (i.e., how many elements it can be accommodate).

```
package main
 import "fmt"
 func main() {
   var numbers = make([]int,3,5)
   printSlice(numbers)
}
func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

append() and copy() Functions

One can increase the capacity of a slice using the append() function. Using copy()function, the contents of a source slice are copied to a destination slice.

```
package main
import "fmt"
 func main()
{
   var numbers []int
   printSlice(numbers)

   /* append allows nil slice */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* add one element to slice*/
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* add more than one element at a time*/
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* create a slice numbers1 with double the capacity of earlier slice*/
```

```
    numbers1 := make([]int, len(numbers), (cap(numbers))*2)

    /* copy content of numbers to numbers1 */
    copy(numbers1,numbers)
    printSlice(numbers1)
}
func printSlice(x []int){
    fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

Output:

```
len = 0 cap = 0 slice = [] len = 1 cap = 2 slice = [0] len = 2 cap = 2 slice = [0 1]
len = 5 cap = 8 slice = [0 1 2 3 4]
len = 5 cap = 16 slice = [0 1 2 3 4]
```

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

```
var map1 map[keytype]valuetype
```

E.g.

```
var map1 map[int]string
```

Example

```
package main
import "fmt"
func main ()  {
x := map[string]int{"Kate":28,"John":37, "Raj":20}fmt.Print(x)
fmt.Println("\n",x["Raj"])
}
```

Output:

```
map[John:37 Raj:20 Kate:28]
 20
```

Insert and Update operation

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 thenupdate operation takes place.

```
package mainimport "fmt" func main() {
m := make(map[string]int)fmt.Println(m)
m["Key1"] = 10
m["Key2"] = 20
m["Key3"] = 30
fmt.Println(m)
m["Key2"] = 555
fmt.Println(m)
}
```

Output:

```
map[]
map[Key3:30 Key1:10 Key2:20] map[Key1:10 Key2:555 Key3:30]
```

Delete operation

You can delete an element in Go Map using delete() functionSyntax:
delete(map, key)

example package mainimport "fmt" func main() {

```
m := make(map[string]int)m["Key1"] = 10
m["Key2"] = 20
m["Key3"] = 30
fmt.Println(m) delete(m, "Key3")fmt.Println(m)
}
```

Output:

```
map[Key1:10 Key2:20 Key3:30]

map[Key2:20 Key1:10]
```