GO :

What is Go?

- 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 known for its simplicity and efficiency
- Go was developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson in 2007
- Go's syntax is similar to C++

What is Go Used For?

- Web development (server-side)
- Developing network-based programs
- Developing cross-platform enterprise applications
- Cloud-native development

Why Use Go?

- Go is fun and easy to learn
- Go has fast run time and compilation time
- Go supports concurrency
- Go has memory management
- Go works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)

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

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

```
C:\Users\tallu\OneDrive\Documents\go>go run helloworld.go
Hello World!
```

Go Syntax

A Go file consists of the following parts:

- Package declaration package main
- Import packages import ("fmt")
- Functions
- Statements and expressions

- Example explained
- **Line 1:** In Go, every program is part of a package. We define this using the `package` keyword. In this example, the program belongs to the `main` package.
- **Line 2:** `import ("fmt")` lets us import files included in the `fmt` package.
- **Line 3:** A blank line. Go ignores white space. Having white spaces in code makes it more readable.
- **Line 4:** `func main() {}` is a function. Any code inside its curly brackets `{}` will be executed.
- **Line 5:** `fmt.Println()` is a function made available from the `fmt` package. It is used to output/print text. In our example it will output "Hello World!".

Go Comments

/* */ ->multi line

// -> single line

Go Variables

- 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

Note: You always have to specify either type or value (or both).

variablename := *value*

```go
package main
import ("fmt")
func main(){
    var s1 string="tnr"
    var a int=5
    fmt.Println(s1)
    fmt.Println(a)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run variables1.go
tnr
5
```

```go
package main
import ("fmt")
func main(){
    a:=5
    fmt.Println(a)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run variables2.go
5
```

```go
package main
import ("fmt")
func main(){
    var a,b,c,d=1,2,3,4
    var e,f=5,"tnr"
    fmt.Println(a,b,c,d,e,f)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run multiple.go
1 2 3 4 5 tnr
```

```go
package main
import ("fmt")
func main(){
    var(
        a=5
        b="tnr"
    )
    fmt.Println(a,b)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run blockvariables.go
5 tnr
```

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

Go Constants

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

const *CONSTNAME type = value*

```go
package main
import ("fmt")
const a int=1//typed constant
const b=2// untyped constant
//constant declared anywhare
func main(){
    const c int=3
    const d=4
    fmt.Println(a,b,c,d)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run constant.go
1 2 3 4
```

Go Output Functions :

- **Print()** function prints its arguments with their default format.
- **Println()** The Println() function is similar to Print() with the difference that a whitespace is added between the arguments, and a newline is added at the end
- **Printf()** The Printf() function first formats its argument based on the given formatting verb and then prints them.

Same like as java

Go format verbs :

Verb	Description
%v	Prints the value in the default format
%#v	Prints the value in Go-syntax format
%T	Prints the type of the value
%%	Prints the % sign

Go Data Types :

- **bool:** represents a boolean value and is either true or false
- **Numeric:** represents integer types, floating point values, and complex types
- **string:** represents a string value

```go
package main
import ("fmt")
func main(){
    var a string="tnr"
    var b int=5
    var c bool=true
    var d float32=31.2
    fmt.Println(a,b,c,d)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run datatype.go
tnr 5 true 31.2
```

- **Signed integers - can store both positive and negative values**
- **Unsigned integers - can only store non-negative values**

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

Type	Size	Range
`int`	Depends on platform: 32 bits in 32 bit systems and 64 bit in 64 bit systems	-2147483648 to 2147483647 in 32 bit systems and -9223372036854775808 to 9223372036854775807 in 64 bit systems
`int8`	8 bits/1 byte	-128 to 127
`int16`	16 bits/2 byte	-32768 to 32767
`int32`	32 bits/4 byte	-2147483648 to 2147483647
`int64`	64 bits/8 byte	-9223372036854775808 to 9223372036854775807

Type	Size	Range
float32	32 bits	-3.4e+38 to 3.4e+38.
float64	64 bits	-1.7e+308 to +1.7e+308.

Go Arrays

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

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

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

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

```go
package main
import ("fmt")
func main(){
    var a=[3]int{1,2,3}
    var b=[...]int{4,5,6}
    c:=[3]int{7,8,9}
    d:=[...]int{10,11,12}
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run arrays.go
[1 2 3]
[4 5 6]
[7 8 9]
[10 11 12]
```

Default values is 0

Accessing and updating are do with indexes

1:10 referes to assign index 1 with 10 arr[1]=10

4:20 arr[4]=20

Len() function is used to get length of the function.

Go Slices

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

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

`cap()` function - returns the capacity of the slice (the number of elements the slice can grow or shrink to)

```go
package main
import ("fmt")
func main(){
    s1:=[]int{}
    fmt.Println(len(s1))
    fmt.Println(cap(s1))
    fmt.Println(s1)
    s2:=[]int{1,2,3,4}
    fmt.Println(len(s2))
    fmt.Println(cap(s2))
    fmt.Println(s2)

    //slice from arry
    arr1:=[5]int{10,11,12,13,14}
    s3:=arr1[2:5]
    fmt.Println(s3)

    //using make function
    s4:=make([]int,5,10)//(datatype,size,capacity)
    fmt.Println(len(s4))
    fmt.Println(cap(s4))
    fmt.Println(s4)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run slice.go
0
0
[]
4
4
[1 2 3 4]
[12 13 14]
5
10
[0 0 0 0 0]
```

All are similar to arrays in slices

slice_name = append(*slice_name*, *element1*, *element2*, ...)

slice3 = append(*slice1*, *slice2*...)

copy(*dest*, *src*) Memory Efficiency

```go
package main
import ("fmt")
func main(){
    a:=[]int{1,2,3,4,5}
    fmt.Println(a)
    a=append(a,6)
    fmt.Println(a)
    a=append(a,7,8)
    fmt.Println(a)
    b:=[]int{1,2}
    fmt.Println(b)
    c:=append(a,b...)
    fmt.Println(c)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run slice2.go
[1 2 3 4 5]
[1 2 3 4 5 6]
[1 2 3 4 5 6 7 8]
[1 2]
[1 2 3 4 5 6 7 8 1 2]
```

Operators are similar to cpp

Conditional statements :

```go
package main

import ("fmt")

func main() {
    a := 20
    if (a >= 18) {
        fmt.Println("Eligible for vote.")
    } else {
        fmt.Println("Not eligible.")
    }
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run if.go
Eligible for vote.
```

Switch case:

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

- The expression is evaluated once
- The value of the switch expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The default keyword is optional. It specifies some code to run if there is no case match

```go
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")
    }
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run switch.go
Thursday
```

Multiple switch case:

switch *expression* {
case *x*,*y*:
  // code block if expression is evaluated to x or y
case *v*,*w*:
  // code block if expression is evaluated to v or w
case *z*:
...
default:
  // code block if expression is not found in any cases
}

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

func main() {
    day := 5

    switch day {
    case 1,3,5:
     fmt.Println("Odd weekday")
    case 2,4:
      fmt.Println("Even weekday")
    case 6,7:
     fmt.Println("Weekend")
   default:
     fmt.Println("Invalid day of day number")
   }
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run multiswitchcase.go
Odd weekday
```

GO LOOPS :

For loop :

for *statement1; statement2; statement3* {

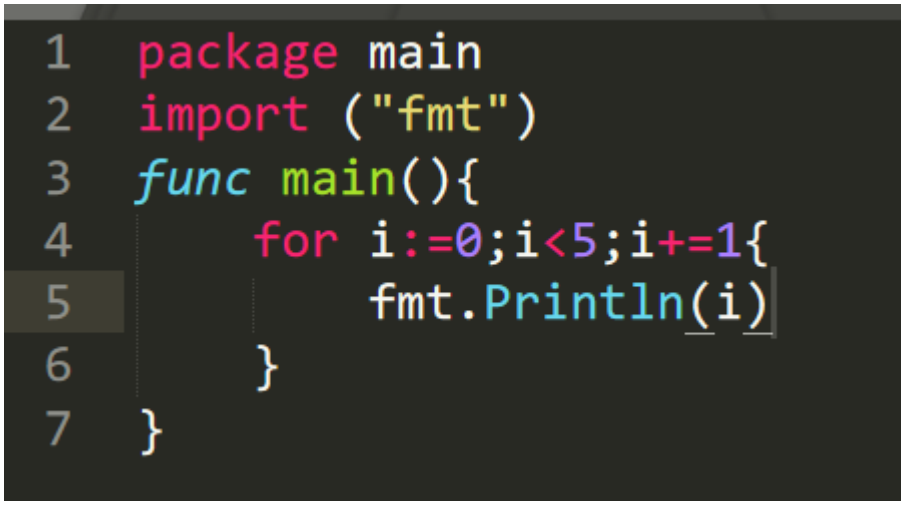
  // code to be executed for each iteration

}

statement1 Initializes the loop counter value.

statement2 Evaluated for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop ends.

statement3 Increases the loop counter value.



```go
package main
import ("fmt")
func main(){
    for i:=0;i<5;i+=1{
        fmt.Println(i)
    }
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run forloop.go
0
1
2
3
4
```

Continue,break,nested loop are same

Go Functions

Create a Function

To create (often referred to as declare) a function, do the following:

- 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 {}.

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

```
func FunctionName(param1 type, param2 type, param3 type) {
  // code to be executed
}
```

```go
package main
import ("fmt")
func mesg(){
    fmt.Println("hi,i am tnr")
}
func sum(a int,b int) int {
    return a+b
}
func pro(a int,b int){
    fmt.Println(a*b)
}
func main(){
    mesg()
    fmt.Println(sum(3,2))
    pro(3,2)
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run functions.go
hi,i am tnr
5
6
```

```go
package main
import ("fmt")
func factorial(n int) int{
    if n==0|| n==1{
        return 1
    } else{
        return n*factorial(n-1)
    }
}
func main(){
    a:=5
    fmt.Println(factorial(a))
}
```

```
C:\Users\tallu\OneDrive\Documents\go>go run factorial.go
120
```

Go Struct

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.

type *struct_name* struct {

member1 datatype;

member2 datatype;

member3 datatype;

...

}

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

func main() {
    var pers1 Person
    var pers2 Person

    // Pers1 specification
    pers1.name = "Hege"
    pers1.age = 45
    pers1.job = "Teacher"
    pers1.salary = 6000

    // Pers2 specification
    pers2.name = "Cecilie"
    pers2.age = 24
    pers2.job = "Marketing"
    pers2.salary = 4500

    // Access and print Pers1 info
    fmt.Println("Name: ", pers1.name)
    fmt.Println("Age: ", pers1.age)
    fmt.Println("Job: ", pers1.job)
    fmt.Println("Salary: ", pers1.salary)
```

```
C:\Users\tallu\OneDrive\Documents\go>go run struct.go
Name:  Hege
Age:  45
Job:  Teacher
Salary:  6000
Name:  Cecilie
Age:  24
Job:  Marketing
Salary:  4500
```

Go Maps:

Go Maps

Maps are used to store data values in key:value pairs.

Each element in a map is a key:value pair.

A map is an unordered and changeable collection that does not allow duplicates.

The length of a map is the number of its elements. You can find it using the len() function.

The default value of a map is nil.

Maps hold references to an underlying hash table.

Go has multiple ways for creating maps.

var *a* = map[KeyType]ValueType{*key1*:*value1*, *key2*:*value2*,...}

b := map[KeyType]ValueType{*key1*:*value1*, *key2*:*value2*,...}

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

func main() {
  var a = map[string]string{"brand": "Ford", "model": "Mustang", "year": "1964"}
  b := map[string]int{"Oslo": 1, "Bergen": 2, "Trondheim": 3, "Stavanger": 4}

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

```
C:\Users\tallu\OneDrive\Documents\go>go run maps.go
map[brand:Ford model:Mustang year:1964]
map[Bergen:2 Oslo:1 Stavanger:4 Trondheim:3]
```