GoLang

Go is a statically typed, concurrent, and garbage-collected programming language created at Google.

Go is known for its support for concurrency, which is the ability to run multiple tasks simultaneously.

Concurrency is achieved in Go through the use of Goroutines and Channels, which allow you to write code that can run multiple operations at the same time.

Makes Go an ideal choice for building high-performance and scalable network services, as well as for solving complex computational problems.

Go is its garbage collection, which automatically manages memory for you. This eliminates the need for manual memory management, reducing the likelihood of memory leaks and other bugs that can arise from manual memory management.

```go
package main
import "fmt"

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

package main of the program, which have overall content of the program.It is the initial point to run the program, So it is compulsory to write.

import "fmt", it is a preprocessor command which tells the compiler to include the files lying in the package.

main function, it is beginning of execution of program.

fmt.Println() is a standard library function to print something as a output on screen.In this, *fmt* package has transmitted Println method which is used to display the output.

Identifiers in Go Language

Identifiers are the user-defined names of the program components. In the Go language, an identifier can be a variable name, function name, constant, statement label, package name, or type.

```go
package main
import "fmt"

func main() {
```

```
 var name = "GeeksforGeeks"
}
```

```
For Constants:                                                    go
true, false, iota, nil

For Types:
int, int8, int16, int32, int64, uint,
uint8, uint16, uint32, uint64, uintptr,
float32, float64, complex128, complex64,
bool, byte, rune, string, error

For Functions:
make, len, cap, new, append, copy, close,
delete, complex, real, imag, panic, recover
```

Data Types in Go

Go language, the type is divided into four categories which are as follows:

1. **Basic type:** Numbers, strings, and booleans come under this category.
2. **Aggregate type:** Array and structs come under this category.
3. **Reference type:** Pointers, slices, maps, functions, and channels come under this category.
4. **Interface type**

The ***Basic Data Types*** are further categorized into three subcategories which are:

- **Numbers**
- **Booleans**
- **Strings**

Data Type	Description
int8	8-bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
int64	64-bit signed integer
uint8	8-bit unsigned integer
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
int	Both int and uint contain same size, either 32 or 64 bit.

uint	Both int and uint contain same size, either 32 or 64 bit.
rune	It is a synonym of int32 and also represent Unicode code points.
byte	It is a synonym of uint8.
uintptr	It is an unsigned integer type. Its width is not defined, but its can hold all the bits of a pointer value.

```go
package main

import "fmt"

func main() {

    var x int16 = 170
    var y int16 = 83
    //Addition
    fmt.Printf(" addition :  %d + %d = %d\n ", x, y, x+y)
    //Subtraction
    fmt.Printf("subtraction : %d - %d = %d\n", x, y, x-y)
    //Multiplication
    fmt.Printf(" multiplication : %d * %d = %d\n", x, y, x*y)
    //Division
    fmt.Printf(" division : %d / %d = %d\n", x, y, x/y)
    //Modulus
    fmt.Printf(" remainder : %d %% %d = %d\n", x, y, x%y)
}
```

Floating-Point Numbers: In Go language, floating-point numbers are divided into **two** categories

Data Type	Description
float32	32-bit IEEE 754 floating-point number
float64	64-bit IEEE 754 floating-point number

```go
package main
import "fmt"

func main() {
    a := 20.45
    b := 34.89

    // Subtraction of two
    // floating-point number
    c := b-a

    // Display the result
    fmt.Printf("Result is: %f", c)

    // Display the type of c variable
```

```
    fmt.Printf("\nThe type of c is : %T", c)
```

```
#Output
Result is: 14.440000
The type of c is : float64
```

Complex Numbers

- There are few built-in functions in complex numbers:
  - complex – make complex numbers from two floats.
  - real() – get real part of the input complex number as a float number.
  - imag() – get imaginary of the input complex number part as float number

Data Type	Description
complex64	Complex numbers which contain float32 as a real and imaginary component.
complex128	Complex numbers which contain float64 as a real and imaginary component.

```go
package main

import "fmt"

func main() {
    comp1 := complex(10, 11)
    // complex number init syntax
    comp2 := 13 + 33i
    fmt.Println("Complex number 1 is :", comp1)
    fmt.Println("Complex number 1 is :", comp2)
    // get real part
    realNum := real(comp1)
    fmt.Println("Real part of complex number 1:", realNum)
    // get imaginary part
    imaginary := imag(comp2)
    fmt.Println("Imaginary part of complex number 2:", imaginary)

}
```

Booleans

The boolean data type represents only one bit of information either true or false. The values of type boolean are not converted implicitly or explicitly to any other type.

```go
package main
import "fmt"

func main() {

    // variables
```

```
    str1 := "GeeksforGeeks"
    str2:= "geeksForgeeks"
    str3:= "GeeksforGeeks"
    result1:= str1 == str2
    result2:= str1 == str3

    // Display the result
    fmt.Println( result1)
    fmt.Println( result2)

    // Display the type of
    // result1 and result2
    fmt.Printf("The type of result1 is %T and "+
                    "the type of result2 is %T",
                              result1, result2)

}
```

Strings

The string data type represents a sequence of Unicode code points.

Strings can be concatenated using plus(+) operator.

```go
package main
import "fmt"

func main() {

    // str variable which stores strings
    str := "GeeksforGeeks"

    // Display the length of the string
    fmt.Printf("Length of the string is:%d",
                                  len(str))

    // Display the string
    fmt.Printf("\nString is: %s", str)

    // Display the type of str variable
    fmt.Printf("\nType of str is: %T", str)
}
```

Go Variables

When a user enters a new value that will be used in the process of operation, can store temporarily in the Random Access Memory of the computer and these values in this part of memory vary throughout the execution and hence another term for this came which is known as ***Variables***.

Constants

CONSTANTS suggests, it means fixed. In programming languages also it is same i.e., once the value of constant is defined, it cannot be modified further.

```go
package main

import "fmt"

const PI = 3.14

func main()
{
    const GFG = "GeeksforGeeks"
    fmt.Println("Hello", GFG)

    fmt.Println("Happy", PI, "Day")

    const Correct= true
    fmt.Println("Go rules?", Correct)
}
```

Untyped and Typed Numeric Constants:

Typed constants work like immutable variables can inter-operate only with the same type and untyped constants work like literals can inter-operate with similar types. Constants can be declared with or without a type in Go.

```go
const untypedInteger          = 123
const untypedFloating          = 123.12

const typedInteger  int             = 123
const typedFloatingPoint   float64  = 123.12
```

list of constants in Go Language:

- Numeric Constant (Integer constant, Floating constant, Complex constant)
- String literals
- Boolean Constant

Numeric Constant: Numeric constants are ***high-precision values***.You can't add a ***float64*** to an ***int***, or even an ***int32*** to an ***int***.

Numeric constant can be of 3 kinds:

Integer Constant:

- It can be a *decimal, octal, or hexadecimal constant*.
- An int can store at maximum a 64-bit integer, and sometimes less.

Complex constant:

Complex constants behave a lot like floating-point constants. It is an *ordered pair* or *real pair* of integer constant(or parameter). And the constant are separated by a comma, and the pair is enclosed in between parentheses.

Floating Type Constant:

- A floating type constant has an *integer part, a decimal point, a fractional part, and an exponent part*.
- *While* represented using the decimal form, it must include the decimal point, the exponent, or both.

Operators

Operators are the foundation of any programming language. Thus the functionality of the Go language is incomplete without the use of operators

- **Arithmetic Operators**
- **Relational Operators**
- **Logical Operators**
- **Bitwise Operators**
- **Assignment Operators**
- **Misc Operators**

Arithmetic Operator

These are used to perform arithmetic/mathematical operations on operands in Go language:

- **Addition:** The '+' operator adds two operands. For example, x+y.
- **Subtraction:** The '-' operator subtracts two operands. For example, x-y.
- **Multiplication:** The '*' operator multiplies two operands. For example, x*y.
- **Division:** The '/' operator divides the first operand by the second. For example, x/y.
- **Modulus:** The '%' operator returns the remainder when the first operand is divided by the second. For example, x%y.

Relational Operator

- **'=='(Equal To)** operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise, it returns false. For example, 5==5 will return true.
- **'!='(Not Equal To)** operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise, it returns false. It is the exact boolean complement of the '==' operator. For example, 5!=5 will return false.
- **'>'(Greater Than)**operator checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise, it returns false. For example, 6>5 will return true.
- **'<'(Less Than)**operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise, it returns false. For example, 6<5 will return false.

- **‘>='(Greater Than Equal To)**operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true. Otherwise, it returns false. For example, 5>=5 will return true.
- **‘<='(Less Than Equal To)**operator checks whether the first operand is lesser than or equal to the second operand. If so, it returns true. Otherwise, it returns false. For example, 5<=5 will also return true

Logical Operators

- **Logical AND:** The ‘&&’ operator returns true when both the conditions in consideration are satisfied. Otherwise it returns false. For example, a && b returns true when both a and b are true (i.e. non-zero).
- **Logical OR:** The ‘||’ operator returns true when one (or both) of the conditions in consideration is satisfied. Otherwise it returns false. For example, a || b returns true if one of a or b is true (i.e. non-zero). Of course, it returns true when both a and b are true.
- **Logical NOT:** The ‘!’ operator returns true the condition in consideration is not satisfied. Otherwise it returns false. For example, !a returns true if a is false, i.e. when a=0.

```go
package main
import "fmt"
func main() {
    var p int = 23
    var q int = 60

    if(p!=q && p<=q){
        fmt.Println("True")
    }

    if(p!=q || p<=q){
        fmt.Println("True")
    }

    if(!(p==q)){
        fmt.Println("True")
    }
}
```

Bitwise Operator

- **& (bitwise AND):** Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- **| (bitwise OR):** Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.
- **^ (bitwise XOR):** Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- **<< (left shift):** Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

- **>> (right shift):** Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- **&^ (AND NOT):** This is a bit clear operator.

```go
package main

import "fmt"

func main() {
   p:= 34
   q:= 20

   // & (bitwise AND)
   result1:= p & q
   fmt.Printf("Result of p & q = %d", result1)

   // | (bitwise OR)
   result2:= p | q
   fmt.Printf("\nResult of p | q = %d", result2)

   // ^ (bitwise XOR)
   result3:= p ^ q
   fmt.Printf("\nResult of p ^ q = %d", result3)

   // << (left shift)
   result4:= p << 1
   fmt.Printf("\nResult of p << 1 = %d", result4)

   // >> (right shift)
   result5:= p >> 1
   fmt.Printf("\nResult of p >> 1 = %d", result5)

   // &^ (AND NOT)
   result6:= p &^ q
   fmt.Printf("\nResult of p &^ q = %d", result6)
}
```

```go
Result of p & q = 0
Result of p | q = 54
Result of p ^ q = 54
Result of p << 1 = 68
Result of p >> 1 = 17
Result of p &^ q = 34
```

Assignment Operators

- **“=”(Simple Assignment):** This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.
- **“+=”(Add Assignment):** This operator is a combination of ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

- **“-=”(Subtract Assignment):** This operator is a combination of ‘-‘ and ‘=’ operators. This operator first subtracts the current value of the variable on left from the value on the right and then assigns the result to the variable on the left.
- **“*=”(Multiply Assignment):** This operator is a combination of ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
- **“/=”(Division Assignment):** This operator is a combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
- **“%=”(Modulus Assignment):** This operator is a combination of ‘%’ and ‘=’ operators. This operator first modulo the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
- **“&=”(Bitwise AND Assignment):** This operator is a combination of ‘&’ and ‘=’ operators. This operator first “Bitwise AND” the current value of the variable on the left by the value on the right and then assigns the result to the variable on the left.
- **“^=”(Bitwise Exclusive OR):** This operator is a combination of ‘^’ and ‘=’ operators. This operator first “Bitwise Exclusive OR” the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
- **“|=”(Bitwise Inclusive OR):** This operator is a combination of ‘|’ and ‘=’ operators. This operator first “Bitwise Inclusive OR” the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
- **“<<=”(Left shift AND assignment operator):** This operator is a combination of ‘<<’ and ‘=’ operators. This operator first “Left shift AND” the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
- **“>>=”(Right shift AND assignment operator):** This operator is a combination of ‘>>’ and ‘=’ operators. This operator first “Right shift AND” the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.

Misc Operator

- **&:** This operator returns the address of the variable.
- ***:** This operator provides pointer to a variable.
- **<-:**The name of this operator is receive. It is used to receive a value from the channel.

```go
package main

import "fmt"

func main() {
  a := 4
  b := &a
  fmt.Println(*b)
  *b = 7
  fmt.Println(a)
}
```

```
4
7
```

if, if-else, Nested-if, if-else-if

Golang uses control statements to control the flow of execution of the program based on certain conditions

if Statement

This is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

```go
if condition {

   // Statements to execute if
   // condition is true
}
```

if…else Statement

if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement.

```go
if condition {

    // Executes this block if
    // condition is true
} else {

    // Executes this block if
    // condition is false
}
```

Nested if Statement

Nested if statements mean an if statement inside an if statement.

```go
if condition1 {

   // Executes when condition1 is true

   if condition2 {

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

if..else..if ladder

one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```go
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 none
     // of the condition is true
}
```

Loops in Go Language

Go language contains only a single loop that is for-loop.

```go
for initialization; condition; post{
       // statements....
}
```

- The *initialization* statement is optional and executes before for loop starts. The initialization statement is always in a simple statement like variable declarations, increment or assignment statements, or function calls.
- The *condition* statement holds a boolean expression, which is evaluated at the starting of each iteration of the loop. If the value of the conditional statement is true, then the loop executes.
- The *post* statement is executed after the body of the for-loop. After the post statement, the condition statement evaluates again if the value of the conditional statement is false, then the loop ends.

```go
package main

import "fmt"

// Main function
func main() {

    // for loop
    // This loop starts when i = 0
```

```
    // executes till i<4 condition is true
    // post statement is i++
    for i := 0; i < 4; i++{
      fmt.Printf("GeeksforGeeks\n")
    }

}
```

For loop as Infinite Loop:

A for loop is also used as an infinite loop by removing all the three expressions from the for loop.

```
for{                                                                    go
     // Statement...
}
```

```
package main                                                            go

import "fmt"

// Main function
func main() {

    // Infinite loop
    for {
      fmt.Printf("GeeksforGeeks\n")
    }

}
```

for loop as while Loop: A for loop can also work as a while loop. This loop is executed until the given condition is true.

```
for condition{                                                          go
    // statement..
}
```

```
package main                                                            go

import "fmt"

// Main function
func main() {

    // while loop
    // for loop executes till
    // i < 3 condition is true
    i:= 0
    for i < 3 {
```

```
        i += 2
    }
  fmt.Println(i)
}
```

Simple range in for loop:

```go
for i, j:= range rvariable{
   // statement..
}
```

```go
package main

import "fmt"

// Main function
func main() {

    // Here rvariable is a array
    rvariable:= []string{"GFG", "Geeks", "GeeksforGeeks"}

    // i and j stores the value of rvariable
    // i store index number of individual string and
    // j store individual string of the given array
    for i, j:= range rvariable {
       fmt.Println(i, j)
    }
}
```

```go
0 GFG
1 Geeks
2 GeeksforGeeks
```

Using for loop for strings: A for loop can iterate over the Unicode code point for a string.

```go
package main

import "fmt"

// Main function
func main() {

    // String as a range in the for loop
    for i, j:= range "XabCd" {
       fmt.Printf("The index number of %U is %d\n", j, i)
    }

}
```

```
The index number of U+0058 is 0
The index number of U+0061 is 1
The index number of U+0062 is 2
The index number of U+0043 is 3
The index number of U+0064 is 4
```

For Maps: A for loop can iterate over the key and value pairs of the map.

```
for key, value := range map {
     // Statement..
}
```

```
package main

import "fmt"

// Main function
func main() {

    // using maps
    mmap := map[int]string{
        22:"Geeks",
        33:"GFG",
        44:"GeeksforGeeks",
    }
    for key, value:= range mmap {
       fmt.Println(key, value)
    }

}
```

```
22 Geeks
33 GFG
44 GeeksforGeeks
```

For Channel: A for loop can iterate over the sequential values sent on the channel until it closed.

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

```
package main

import "fmt"

// Main function
func main() {
```

```
    // using channel
    chnl := make(chan int)
    go func(){
        chnl <- 100
        chnl <- 1000
       chnl <- 10000
       chnl <- 100000
       close(chnl)
    }()
    for i:= range chnl {
       fmt.Println(i)
    }

}
```

```
100
1000
10000
100000
```
go

Switch Statement in Go

A switch statement is a multiway branch statement. It provides an efficient way to transfer the execution to different parts of a code based on the value(also called case) of the expression.

1. **Expression Switch**
2. **Type Switch**

Expression Switch

Expression switch is similar to switch statement in C, C++, Java language. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

```
switch optstatement; optexpression{
case expression1: Statement..
case expression2: Statement..
...
default: Statement..
}
```
go

```
package main

import "fmt"

func main() {
```
go

```
    // Switch statement with both
    // optional statement, i.e, day:=4
    // and expression, i.e, day
    switch day:=4; 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")
       default:
       fmt.Println("Invalid")
   }

}
```

```go
package main

import "fmt"

func main() {
    var value int = 2

    // Switch statement without an
    // optional statement and
    // expression
   switch {
       case value == 1:
       fmt.Println("Hello")
       case value == 2:
       fmt.Println("Bonjour")
       case value == 3:
       fmt.Println("Namstay")
       default:
       fmt.Println("Invalid")
   }

}
```

```go
package main

import "fmt"

func main() {
```

```
    var value string = "five"

    // Switch statement without default statement
    // Multiple values in case statement
   switch value {
       case "one":
       fmt.Println("C#")
       case "two", "three":
       fmt.Println("Go")
       case "four", "five", "six":
       fmt.Println("Java")
   }
}
```

Type Switch

Type switch is used when you want to compare types. In this switch, the case contains the type which is going to compare with the type present in the switch expression

```
switch optstatement; typeswitchexpression{
case typelist 1: Statement..
case typelist 2: Statement..
...
default: Statement..
}
```
go

```
package main

import "fmt"

func main() {
    var value interface{}
    switch q:= value.(type) {
       case bool:
       fmt.Println("value is of boolean type")
       case float64:
       fmt.Println("value is of float64 type")
       case int:
       fmt.Println("value is of int type")
       default:
       fmt.Printf("value is of type: %T", q)

   }
}
```
go

```
value is of type: <nil>
```
go

Functions in Go Language

Functions are generally the block of codes or statements in a program that gives the user the ability to reuse the same code which ultimately saves the excessive use of memory, acts as a time saver and more importantly, provides better readability of the code.

```go
func function_name(Parameter-list)(Return_type){
    // function body.....
}
```

- **func:** It is a keyword in Go language, which is used to create a function.
- **function_name:** It is the name of the function.
- **Parameter-list:** It contains the name and the type of the function parameters.
- **Return_type:** It is optional and it contain the types of the values that function returns. If you are using return_type in your function, then it is necessary to use a return statement in your function.

```go
package main
import "fmt"

// area() is used to find the
// area of the rectangle
// area() function two parameters,
// i.e, length and width
func area(length, width int)int{

    Ar := length* width
    return Ar
}

// Main function
func main() {

   // Display the area of the rectangle
   // with method calling
   fmt.Printf("Area of rectangle is : %d", area(12, 10))
}
```

In Go language, the parameters passed to a function are called actual parameters, whereas the parameters received by a function are called formal parameters.

Call by value: : In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations.

```go
package main

import "fmt"
```

```
// function which swap values
func swap(a, b int)int{

    var o int
    o= a
    a=b
    b=o

   return o
}

// Main function
func main() {
 var p int = 10
 var q int = 20
  fmt.Printf("p = %d and q = %d", p, q)

 // call by values
 swap(p, q)
   fmt.Printf("\np = %d and q = %d",p, q)
}
```

```
p = 10 and q = 20
p = 10 and q = 20
```

go

Call by reference: Both the actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in actual parameters of the caller.

```
package main
import "fmt"
// function which swap values
func swap(a, b *int)int{
    var o int
    o = *a
    *a = *b
    *b = o

   return o
}

// Main function
func main() {

 var p int = 10
 var q int = 20
 fmt.Printf("p = %d and q = %d", p, q)

 // call by reference
 swap(&p, &q)
```

go

```
    fmt.Printf("\np = %d and q = %d",p, q)
}
```

```
p = 10 and q = 20
p = 20 and q = 10
```

Variadic Functions in Go

The function that is called with the varying number of arguments is known as variadic function. Or in other words, a user is allowed to pass zero or more arguments in the variadic function.

```
function function_name(para1, para2...type)type {
     // code...
}
```

```

// Go program to illustrate the
// concept of variadic function
package main

import (
    "fmt"
    "strings"
)

// Variadic function to join strings
func joinstr(elements ...string) string {
    return strings.Join(elements, "-")
}

func main() {

    // zero argument
    fmt.Println(joinstr())

    // multiple arguments
    fmt.Println(joinstr("GEEK", "GFG"))
    fmt.Println(joinstr("Geeks", "for", "Geeks"))
    fmt.Println(joinstr("G", "E", "E", "k", "S"))

}
```

Anonymous function in Go Language

Go language provides a special feature known as an anonymous function. An anonymous function is a function which doesn't contain any name. It is useful when you want to create an inline function.

```go
func(parameter_list)(return_type){
// code..

// Use return statement if return_type are given
// if return_type is not given, then do not
// use return statement
return
}()
```

```go
package main

import "fmt"

func main() {

    // Anonymous function
   func(){

      fmt.Println("Welcome! to GeeksforGeeks")
  }()

}
```

```go
package main

import "fmt"

func main() {

    // Assigning an anonymous
   // function to a variable
   value := func(){
      fmt.Println("Welcome! to GeeksforGeeks")
  }
  value()

}
```

Pass arguments in the anonymous function.

```go
package main

import "fmt"

func main() {

    // Passing arguments in anonymous function
  func(ele string){
```

```
      fmt.Println(ele)
  }("GeeksforGeeks")

}
```

Return an anonymous function from another function.

```go
package main

import "fmt"

 // Returning anonymous function
 func GFG() func(i, j string) string{
     myf := func(i, j string)string{
          return i + j + "GeeksforGeeks"
     }
    return myf
 }

func main() {
    value := GFG()
    fmt.Println(value("Welcome ", "to "))
}
```

Main and init function in Golang

Go language reserve two functions for special purpose and the functions are **main()** and **init()** function.

In Go language, the **main** package is a special package which is used with the programs that are executable and this package contains *main()* function.

Go automatically call *main()* function, so there is no need to call *main()* function explicitly and every executable program must contain single main package and *main()* function.

init() function is just like the main function, does not take any argument nor return anything. This function is present in every package and this function is called when the package is initialized.

```go
package main

// Importing package
import "fmt"

// Multiple init() function
func init() {
    fmt.Println("Welcome to init() function")
}

func init() {
```

```
    fmt.Println("Hello! init() function")
}

// Main function
func main() {
    fmt.Println("Welcome to main() function")
}
```

```
Welcome to init() function
Hello! init() function
Welcome to main() function
```

go

What is Blank Identifier(underscore) in Golang?

Identifiers are the user-defined name of the program components used for the identification purpose. Golang has a special feature to define and use the unused variable using Blank Identifier.

Unused variables are those **variables** that are defined by the user throughout the program but he/she never makes use of these variables. These variables make the program almost unreadable

The real use of Blank Identifier comes when a function returns multiple values, but we need only a few values and want to discard some values. Basically, it tells the compiler that this variable is not needed and ignored it without any error

```go
package main

import "fmt"

// Main function
func main() {

    // calling the function
    // function returns two values which are
    // assigned to mul and div identifier
    mul, div := mul_div(105, 7)

    // only using the mul variable
    // compiler will give an error
    fmt.Println("105 x 7 = ", mul)
}

// function returning two
// values of integer type
func mul_div(n1 int, n2 int) (int, int) {

    // returning the values
```

```
    return n1 * n2, n1 / n2
}
```

```
./prog.go:15:7: div declared and not used                                    go
```

```
package main                                                                 go

import "fmt"

// Main function
func main() {

    // calling the function
    // function returns two values which are
    // assigned to mul and blank identifier
    mul, _ := mul_div(105, 7)

    // only using the mul variable
    fmt.Println("105 x 7 = ", mul)
}

// function returning two
// values of integer type
func mul_div(n1 int, n2 int) (int, int) {

    // returning the values
    return n1 * n2, n1 / n2
}
```

```
105 x 7 =  735                                                               go
```

Defer Keyword in Golang

Defer statements delay the execution of the **function** or method or an **anonymous method** until the nearby functions returns.

```
// Function                                                                  go
defer func func_name(parameter_list Type)return_type{
// Code
}

// Method
defer func (receiver Type) method_name(parameter_list){
// Code
}

defer func (parameter_list)(return_type){
// code
}()
```

```go
package main

import "fmt"

// Functions
func mul(a1, a2 int) int {

    res := a1 * a2
    fmt.Println("Result: ", res)
    return 0
}

func show() {
    fmt.Println("Hello!, GeeksforGeeks")
}

// Main function
func main() {

    // Calling mul() function
    // Here mul function behaves
    // like a normal function
    mul(23, 45)

    // Calling mul()function
    // Using defer keyword
    // Here the mul() function
    // is defer function
    defer mul(23, 56)

    // Calling show() function
    show()
}
```

```
Result:  1035
Hello!, GeeksforGeeks
Result:  1288
```

```go
package main

import "fmt"

// Functions
func add(a1, a2 int) int {
    res := a1 + a2
    fmt.Println("Result: ", res)
    return 0
}

// Main function
```

```go
func main() {

    fmt.Println("Start")

    // Multiple defer statements
    // Executes in LIFO order
    defer fmt.Println("End")
    defer add(34, 56)
    defer add(10, 10)
}
```

```
Start
Result:  20
Result:  90
End
```

Methods in Golang

Go methods are similar to Go function with one difference, i.e, the method contains a receiver argument in it. With the help of the receiver argument, the method can access the properties of the receiver.

```go
func(reciver_name Type) method_name(parameter_list)(return_type){
// Code
}
```

Methods with Struct Type Receiver

In Go language, you are allowed to define a method whose receiver is of a struct type.

```go
// Go program to illustrate the
// method with struct type receiver
package main

import "fmt"

// Author structure
type author struct {
    name     string
    branch string
    particles int
    salary int
}

// Method with a receiver
// of author type
func (a author) show() {

    fmt.Println("Author's Name: ", a.name)
    fmt.Println("Branch Name: ", a.branch)
```

```
    fmt.Println("Published articles: ", a.particles)
    fmt.Println("Salary: ", a.salary)
}

// Main function
func main() {

    // Initializing the values
    // of the author structure
    res := author{
        name:    "Sona",
        branch: "CSE",
        particles: 203,
        salary: 34000,
    }

    // Calling the method
    res.show()
}
```

Method with Non-Struct Type Receiver

In Go language, you are allowed to create a method with non-struct type receiver as long as the type and the method definitions are present in the same package.

```
// Go program to illustrate the method                                    go
// with non-struct type receiver
package main

import "fmt"

// Type definition
type data int

// Defining a method with
// non-struct type receiver
func (d1 data) multiply(d2 data) data {
    return d1 * d2
}

/*
// if you try to run this code,
// then compiler will throw an error
func(d1 int)multiply(d2 int)int{
return d1 * d2
}
*/

// Main function
func main() {
    value1 := data(23)
    value2 := data(20)
```

```
    res := value1.multiply(value2)
    fmt.Println("Final result: ", res)
}
```

Methods with Pointer Receiver

In Go language, you are allowed to create a method with a **pointer** receiver. With the help of a pointer receiver, if a change is made in the method, it will reflect in the caller which is not possible with the value receiver methods.

```go
func (p *Type) method_name(...Type) Type {
// Code
}
```

```go
// Go program to illustrate pointer receiver
package main

import "fmt"

// Author structure
type author struct {
    name     string
    branch string
    particles int
}

// Method with a receiver of author type
func (a *author) show(abranch string) {
    (*a).branch = abranch
}

// Main function
func main() {

    // Initializing the values
    // of the author structure
    res := author{
        name: "Sona",
        branch: "CSE",
    }

    fmt.Println("Author's name: ", res.name)
    fmt.Println("Branch Name(Before): ", res.branch)

    // Creating a pointer
    p := &res

    // Calling the show method
    p.show("ECE")
    fmt.Println("Author's name: ", res.name)
```

```
    fmt.Println("Branch Name(After): ", res.branch)
}
```

Method Can Accept both Pointer and Value

When a function has a value argument, then it will only accept the values of the parameter, and if you try to pass a pointer to a value function, then it will not accept and vice versa. But a Go method can accept both value and pointer, whether it is defined with pointer or value receiver.

```go
// Go program to illustrate how the
// method can accept pointer and value

package main

import "fmt"

// Author structure
type author struct {
    name string
    branch string
}

// Method with a pointer
// receiver of author type
func (a *author) show_1(abranch string) {
    (*a).branch = abranch
}

// Method with a value
// receiver of author type
func (a author) show_2() {

    a.name = "Gourav"
    fmt.Println("Author's name(Before) : ", a.name)
}

// Main function
func main() {

    // Initializing the values
    // of the author structure
    res := author{
        name: "Sona",
        branch: "CSE",
    }

    fmt.Println("Branch Name(Before): ", res.branch)

    // Calling the show_1 method
    // (pointer method) with value
    res.show_1("ECE")
```

```
    fmt.Println("Branch Name(After): ", res.branch)

    // Calling the show_2 method
    // (value method) with a pointer
    (&res).show_2()
    fmt.Println("Author's name(After): ", res.name)
}
```

Difference Between Method and Function

Method	Function
It contains a receiver.	It does not contain a receiver.
Methods of the same name but different types can be defined in the program.	Functions of the same name but different type are not allowed to be defined in the program.
It cannot be used as a first-order object.	It can be used as first-order objects and can be passed

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.

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

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

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.

To Define a structure

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

```go
// 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)
}
```

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.

```
// Golang program to illustrate                                    go
// 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)
}
```

Advantages of using structures in Go:

1. Encapsulation: Structures allow you to encapsulate related data together, making it easier to manage and modify the data.
2. Code organization: Structures help to organize code in a logical way, which makes it easier to read and maintain.
3. Flexibility: Structures allow you to define custom types with their own behavior, making it easier to work with complex data.
4. Type safety: Structures provide type safety by allowing you to define the type of each field, which helps to prevent errors caused by assigning the wrong type of value.
5. Efficiency: Structures in Go are very efficient, both in terms of memory usage and performance.

Disadvantages of using structures in Go:

1. Complexity: Structures can make code more complex, especially if the structures have a large number of fields or methods.
2. Boilerplate code: When defining large structures with many fields, it can be time-consuming to write out all of the field names and types.
3. Inheritance: Go does not support inheritance, which can make it more difficult to work with large hierarchies of related data.
4. Immutability: Go structures are mutable by default, which can make it more difficult to enforce immutability in your code.
5. Overall, the advantages of using structures in Go typically outweigh the disadvantages, as they provide a powerful tool for managing and working with complex data. However, as with any programming technique, it’s important to use structures judiciously and be aware of their limitations.

Nested Structure in Golang

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.

```go
type struct_name_1 struct{
  // Fields
}
type struct_name_2 struct{
  variable_name  struct_name_1

}
```

```go
// Golang program to illustrate
// the nested structure
package main

import "fmt"

// Creating structure
type Author struct {
    name string
    branch string
    year int
}

// Creating nested structure
type HR struct {

    // structure as a field
    details Author
}

func main() {

    // Initializing the fields
    // of the structure
    result := HR{

        details: Author{"Sona", "ECE", 2013},
    }

    // Display the values
    fmt.Println("\nDetails of Author")
    fmt.Println(result)
}
```

Anonymous Structure and Field in Golang

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.

```go
variable_name := struct{
// fields
}{// Field_values}
```

Arrays in Go

In an array, you are allowed to store zero or more than zero elements in it.

The elements of the array are indexed by using the [] index operator with their zero-based position, which means the index of the first element is *array[0]* and the index of the last element is *array[len(array)-1]*.

In Go language, arrays are created in two different ways:

Using var keyword: In Go language, an array is created using the var keyword of a particular type with name, size, and elements.

```go
Var array_name[length]Type
```

In Go language, arrays are mutable, so that you can use array[index] syntax to the left-hand side of the assignment to set the elements of the array at the given index.

```
Var array_name[index] = element
```

Using shorthand declaration:

```
array_name:= [length]Type{item1, item2, item3,...itemN}
```

```go
package main

import "fmt"

func main() {

// Shorthand declaration of array
arr:= [4]string{"geek", "gfg", "Geeks1231", "GeeksforGeeks"}

// Accessing the elements of
// the array Using for loop
fmt.Println("Elements of the array:")

for i:= 0; i < 3; i++{
fmt.Println(arr[i])
}
}
```

Multi-Dimensional Array

Multi-Dimensional arrays are the ***arrays of arrays of the same type***.

```
Array_name[Length1][Length2]..[LengthN]Type
```

In an array, if an array does not initialize explicitly, then the **default value of this array is 0**

In Go language, **an array is of value type not of reference type**. So when the array is assigned to a new variable, then the changes made in the new variable do not affect the original array

```go
// Go program to illustrate value type array
package main

import "fmt"

func main() {

// Creating an array whose size
// is represented by the ellipsis
my_array:= [...]int{100, 200, 300, 400, 500}
fmt.Println("Original array(Before):", my_array)

// Creating a new variable
// and initialize with my_array
new_array := my_array

fmt.Println("New array(before):", new_array)

// Change the value at index 0 to 500
new_array[0] = 500

fmt.Println("New array(After):", new_array)

fmt.Println("Original array(After):", my_array)
}
```

We can directly compare two arrays using == operator.

How to Copy an Array into Another Array in Golang?

```go
package main

import "fmt"

func main() {
    my_arr1 := [5]string{"Scala", "Perl", "Java", "Python", "Go"}
    my_arr2 := my_arr1

    fmt.Println("Array_1: ", my_arr1)
    fmt.Println("Array_2:", my_arr2)
```

```
    my_arr1[0] = "C++"
    fmt.Println("\nArray_1: ", my_arr1)
    fmt.Println("Array_2: ", my_arr2)
}
```

```
Array_1:  [Scala Perl Java Python Go]
Array_2: [Scala Perl Java Python Go]

Array_1:  [C++ Perl Java Python Go]
Array_2:  [Scala Perl Java Python Go]
```

How to pass an Array to a Function in Golang?

```go
// For sized array
func function_name(variable_name [size]type){
// Code
}
```

```go
package main

import "fmt"
func myfun(a [6]int, size int) int {
    var k, val, r int

    for k = 0; k < size; k++ {
        val += a[k]
    }

    r = val / size
    return r
}

func main() {
    var arr = [6]int{67, 59, 29, 35, 4, 34}
    var res int
    res = myfun(arr, 6)
    fmt.Printf("Final result is: %d ", res)
}
```

Slices in Golang

Slices in Go are a flexible and efficient way to represent arrays, and they are often used in place of arrays because of their dynamic size and added features.

The first index position in a slice is always 0 and the last one will be (length of slice – 1).

```go
package main

import "fmt"
```

```go
func main() {
    array := [5]int{1, 2, 3, 4, 5}
    slice := array[1:4]

    fmt.Println("Array: ", array)
    fmt.Println("Slice: ", slice)
}
```

```
Array: [1 2 3 4 5]
Slice: [2 3 4]
```

How to add elements to a slice in Go

```go
package main
import "fmt"

func main() {
    slice := []int{1, 2, 3}
    slice = append(slice, 4, 5, 6)

    fmt.Println("Slice: ", slice)
}
```

```
Slice: [1 2 3 4 5 6]
```

Declaration of Slice

A slice is declared just like an array, but it doesn't contain the size of the slice. So it can grow or shrink according to the requirement.

```go
// Golang program to illustrate how to
// create slices from the slice
package main

import "fmt"

func main() {

    // Creating s slice
    oRignAl_slice := []int{90, 60, 40, 50,
        34, 49, 30}

    // Creating slices from the given slice
    var my_slice_1 = oRignAl_slice[1:5]
    my_slice_2 := oRignAl_slice[0:]
    my_slice_3 := oRignAl_slice[:6]
    my_slice_4 := oRignAl_slice[:]
    my_slice_5 := my_slice_3[2:4]

    // Display the result
```

```
    fmt.Println("Original Slice:", oRignAl_slice)
    fmt.Println("New Slice 1:", my_slice_1)
    fmt.Println("New Slice 2:", my_slice_2)
    fmt.Println("New Slice 3:", my_slice_3)
    fmt.Println("New Slice 4:", my_slice_4)
    fmt.Println("New Slice 5:", my_slice_5)
}
```

How to sort a slice of ints in Golang?

Go language allows you to sort the elements of the slice according to its type. So, an int type slice is sorted by using the following functions. These functions are defined under the sort package so, you have to import sort package in your program for accessing these functions:

1. Ints: This function is used to only sorts a slice of ints and it sorts the elements of the slice in increasing order.

```go
package main

import (
    "fmt"
    "sort"
)

func main() {
    intSlice := []int{5, 2, 6, 3, 1, 4}
    sort.Ints(intSlice)
    fmt.Println(intSlice) // [1 2 3 4 5 6]
}
```

2. IntsAreSorted: This function is used to check whether the given slice of ints is in the sorted form(in increasing order) or not.

```go
// Go program to illustrate how to check
// whether the given slice of ints is in
// sorted form or not
package main

import (
    "fmt"
    "sort"
)

// Main function
func main() {

    // Creating and initializing slices
    // Using shorthand declaration
```

```
    scl1 := []int{100, 200, 300, 400, 500, 600, 700}
    scl2 := []int{-23, 567, -34, 67, 0, 12, -5}

    // Displaying slices
    fmt.Println("Slices:")
    fmt.Println("Slice 1: ", scl1)
    fmt.Println("Slice 2: ", scl2)

    // Checking the slice is in sorted form or not
    // Using IntsAreSorted function
    res1 := sort.IntsAreSorted(scl1)
    res2 := sort.IntsAreSorted(scl2)

    // Displaying the result
    fmt.Println("\nResult:")
    fmt.Println("Is Slice 1 is sorted?: ", res1)
    fmt.Println("Is Slice 2 is sorted?: ", res2)
}
```