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(variable types are known at compile time., compiled language
•Go was developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson in 2007

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.)

Features of Go

- Support for environment adopting patterns similar to dynamic languages. For example, type inference (x := 0 is valid declaration of a variable x of type int)
- Fast Compile Time
- InBuilt Concurrency Support

Features excluded In Golang

- Support for method or operator overloading
- Support for circular dependencies among packages
- Support for pointer arithmetic
- Support for assertions
- Support for type inheritance

Env SetUp

Identify the package you want from

https://go.dev/dl/

In Linux

We should choose the required archive file for installation. For Example, if we are installing Go version 1.20.6 for 64-bit x86 on Linux, the archive would be **go1.20.6.linux-amd64.tar.gz**

Now download the archive and extract it in /usr/local directory. We need to create a Go tree in /usr/local/go directory through following command:

```
tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

To set path, add /usr/local/go/bin to the PATH environment variable. We can do this by adding following line to the command line:

```
export PATH=$PATH:/usr/local/go/bin
```

In Windows

Choose the required archive file for Windows installation.

Download msi file and run installation process.

GOROOT and GOPATH

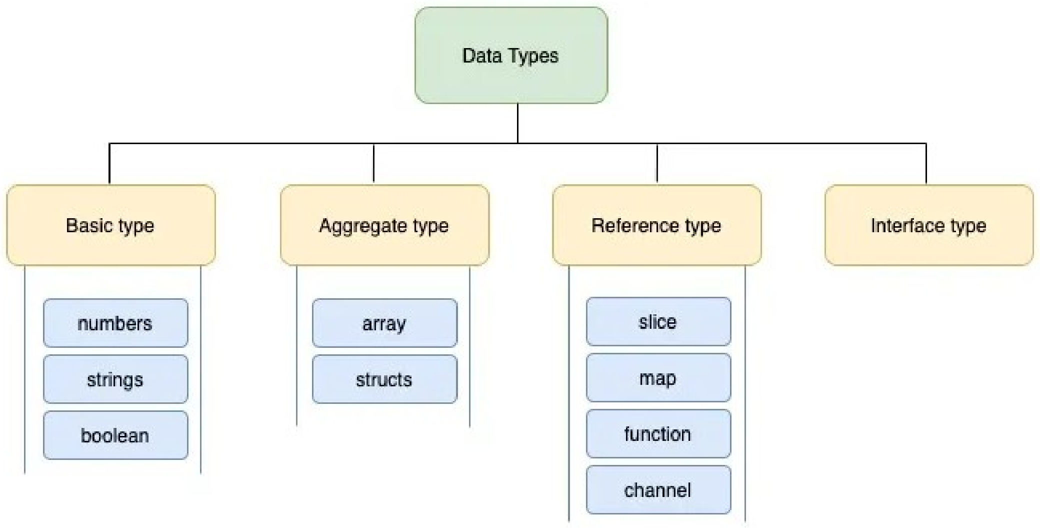
If you have choosed a different directory other than c:\Go, you must set the GOROOT environment variable to your chosen path.

Add the bin subdirectory of your Go root (for example, c:\Go\bin) to your PATH environment variable.

Basic Syntax

```
package main

import "fmt"

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

Data Types



Variables

- 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
- int- stores integers (whole numbers), such as 123 or -123

Declaring Variables

1. With var keyword
 var variablename type = value

 2. With :=
 variablename := value

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

Camel Case : myVariable
Pascal Case:MyVariable
SnakeCase : my_variable

```
package main
import ("fmt")

func main() {
  var student1 string
  var student2 = "Acts" //type is inferred
  x := 20 //type is inferred
  Student1 = "CDAC"

  fmt.Println(student1)
  fmt.Println(student2)
  fmt.Println(x)

  var a, b, c, d int = 1, 3, 5, 7

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

   var (
     a1 int
     b1 int = 1
     c1 string = "hello"
   )

  fmt.Println(a1)
  fmt.Println(b1)
  fmt.Println(c1)
}
```

Constants

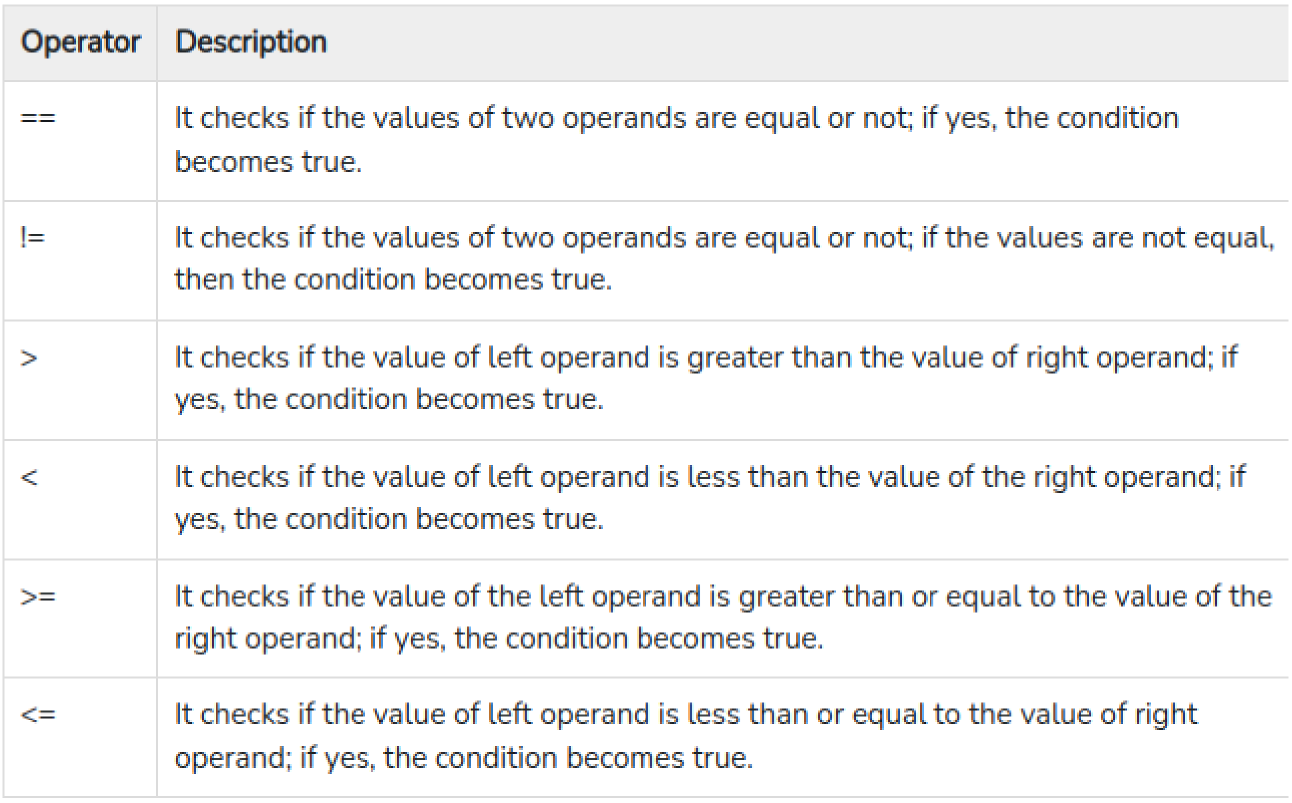
```
package main
import ("fmt")

const PI = 3.14

func main() {
  fmt.Println(PI)
}
```

Operators

1. Arithmatic

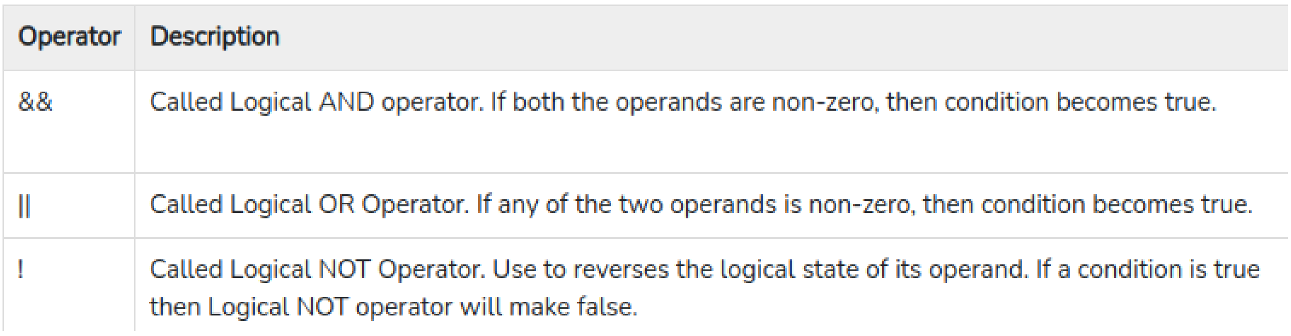
   +, -, ++, --, *, /, %

2. Relational



Operator	Description
==	It checks if the values of two operands are equal or not; if yes, the condition becomes true.
!=	It checks if the values of two operands are equal or not; if the values are not equal, then the condition becomes true.
>	It checks if the value of left operand is greater than the value of right operand; if yes, the condition becomes true.
<	It checks if the value of left operand is less than the value of the right operand; if yes, the condition becomes true.
>=	It checks if the value of the left operand is greater than or equal to the value of the right operand; if yes, the condition becomes true.
<=	It checks if the value of left operand is less than or equal to the value of right operand; if yes, the condition becomes true.

3. Logical



Operator	Description
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

4. Bitwise

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

5. Assignment

Operator	Description
=	Simple assignment operator, Assigns values from right side operands to left side operand
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand
<<=	Left shift AND assignment operator
>>=	Right shift AND assignment operator
&=	Bitwise AND assignment operator
^=	bitwise exclusive OR and assignment operator
|=	bitwise inclusive OR and assignment operator

Loops

In GoLang only for loop is available

```
package main
import ("fmt")

func main() {
  fruits := [3]string{"apple", "orange", "banana"}

    for j:=0; j < len(fruits); j++ {
      fmt.Println(fruits[j])
    }

    for _, val := range fruits {
     fmt.Printf("%v\n", val)
  }

}
```

Array & Slice

The basic difference between a slice and an array is that a slice is a reference to a contiguous segment of an array.

Array which is a value-type and slice is a reference type.

A slice can be a complete array or a part of an array, indicated by the start and end index

Array can store a fixed-size sequential collection of elements of the same type

Slice increases or reduces size dynamically

// Array Declaration

var variable_name [SIZE] variable_type, If you omit the size of the array, an array just big enough to hold the initialization is created.
var variable_name = []variable_type{val1, val2, val3}

// Slice Declaration

var variable_name = []variable_type

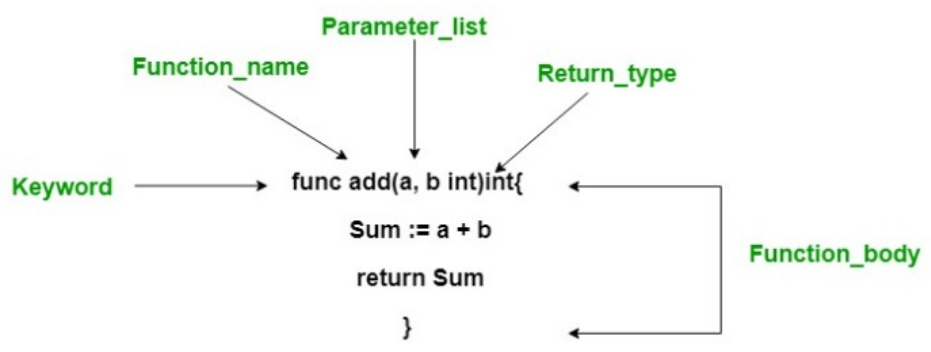
variable_name = make([]variable_type,lenghth,capacity)
length : current length of slice
capacity: maximum number of elements that it can accomodate

```
package main

import "fmt"
func main() {
   var n [10]int /* n is an array of 10 integers */
   var i,j int

   /* initialize elements of array n to 0 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* set element at location i to i + 100 */
   }
   /* output each array element's value */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
   }

    // nil slice
    var numbers []int
    fmt.Printf("len=%d cap=%d slice=%v\n",len(numbers),cap(numbers),
numbers)
     var numbers = make([]int,3,5)
  fmt.Printf("len=%d cap=%d slice=%v\n",len(numbers),cap(numbers),
numbers)

    odd := [6]int{2, 4, 6, 8, 10, 12}
    var s []int = odd[1:4]
    fmt.Println(s)

}
```

Functions

Basic Syntax:

```
   func function_name(parameter parameter_type)(return
return_type){

       // Function Body

}
```



```
package main

import "fmt"

// Basic function taking 2 int args and returning only one value
func area(length, width int)int{

   Ar := length* width
   return Ar
}

//Basic function taking 2 string args and returning two values
func swap(x, y string) (string, string) {
  return y, x
}

func main() {

   fmt.Println(area(5,2))
   fmt.Println(swap("CADC","ACTS"))
}
```

Types Of Functions

1. Variadic Functions

The function that is called with the varying number of arguments is known as variadic function

fmt.Printf is an example of the variadic function.

In the declaration of the variadic function, the type of the last parameter is preceded by an ellipsis, i.e, (**...**). It indicates that the function can be called with any number of parameters of this type.

Basic Syntax:

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

When to Use?

- •Variadic function is used when we don’t know the number of parameters.
- • Variadic functions can be used instead of passing a slice to a function.
- •When you use variadic function in your program, it increases the readability of your program.
- •Variadic functions are especially useful when the arguments to your function are most likely not going to come from a single data structure

The inbuilt append() function is an example of a variadic function.

2. Anonymous Functions

An anonymous function is a function which doesn’t contain any name. It is useful when you want to create an inline function. In Go language, an anonymous function can form a closure. An anonymous function is also known as *function literal*.

Basic Syntax:

```
func(parameter_list)(return_type){

// code..

return

}()
```

3. Methods

Function with a receiver arguement.

the method can access the properties of the receiver.

When you create a method in your code the receiver and receiver type must be present in the same package.

Basic Syntax:

```
func(reciver_name Type) method_name(parameter_list)
(return_type){

// Code

}
```

4. Defer

defer statements delay the execution of the functionor method or an anonymous method until the nearby functions returns.

In other words, defer function or method call arguments evaluate instantly, but they don’t execute until the nearby functions returns.

You can create a deferred method, or function, or anonymous function by using the defer keyword.

multiple defer statements are allowed in the same program and they are executed in LIFO(Last-In, First-Out)

In the defer statements, the arguments are evaluated when the defer statement is executed, not when it is called.

Defer statements are generally used to ensure that the files are closed when their need is over, or to close the channel, or to close any open database connections, or to catch the panics in the program.

Basic Syntax:

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

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

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

```go
package main

import (
    "fmt"
    "strings"
)

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

// Variadic function to return sum of integers
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {
    // pass a slice in variadic function
    elements := []string{"CDAC", "ACTS", "DHPCAP"}
    fmt.Println(joinstr(elements...))
    fmt.Println(joinstr("CDAC", "ACTS", "DHPCAP"))
    sum(4,5,6)
    sum(4,5,6,7,8,9)

}
```

```
package main

import "fmt"


func findSquare(num int) int {
  square := num * num
  return square
}


func main() {

   func(){

      fmt.Println("Welcome to CDAC ACTS!")
  }()

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

 // anonymous function with arguments
  var sum = func(n1, n2 int) {
    sum := n1 + n2
    fmt.Println("Sum is:", sum)
  }
  sum(5, 3)

//return value from an anonymous function
  var sum1 = func(n1, n2 int) int {
    sum1 := n1 + n2
    return sum1
  }

  result := sum1(5, 3)
  fmt.Println("Sum is:", result)

area := func(length, breadth int) int {
    return length * breadth
  }

  fmt.Println("The area of rectangle is", area(3,4))

// Anonymous function as an arguement to another function
  sum2 := func(number1 int, number2 int) int {
    return number1 + number2
}

  result := findSquare(sum2(6, 9))
  fmt.Println("Result is:", result)

}
```

```
package main

import "fmt"

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

// Method with a receiver
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)
}

type Mutatable struct {
        a int
        b int
}

func (m Mutatable) StayTheSame() {
        m.a = 5
        m.b = 7
}

func (m *Mutatable) Mutate() {
        m.a = 5
        m.b = 7
}

func main() {
        res := author{
                name:      "Sona",
                branch:    "CSE",
                particles: 203,
                salary:    34000,
        }

         res.show()
        m := &Mutatable{9, 0}
        fmt.Println(m)
        m.StayTheSame()
        fmt.Println(m)
        m.Mutate()
        fmt.Println(m)
}
```

```
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 from Show Function")
}
func add(a1, a2 int) int {
	res := a1 + a2
	fmt.Println("Result: ", res)
	return 0
}

func main() {
	mul(23, 45)

	defer func() {
	fmt.Println("Defer1")

	}()

	defer mul(23, 56)

	show()
	fmt.Println("Start")
	defer func() {
	fmt.Println("Defer2")

	}()

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

Structure(Struct)

Struct is a user-defined type that allows to group/combine items of possibly different types into a single type.

```
package main
import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books    /* Declare Book1 of type Book */
   var Book2 Books    /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "Mahesh Kumar"
   Book1.subject = "Go Programming Tutorial"
   Book1.book_id = 6495407

   /* book 2 specification */
   Book2.title = "Telecom Billing"
   Book2.author = "Zara Ali"
   Book2.subject = "Telecom Billing Tutorial"
   Book2.book_id = 6495700

   /* print Book1 info */
   fmt.Printf( "Book 1 title : %s\n", Book1.title)
   fmt.Printf( "Book 1 author : %s\n", Book1.author)
   fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
   fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)

   /* print Book2 info */
   fmt.Printf( "Book 2 title : %s\n", Book2.title)
   fmt.Printf( "Book 2 author : %s\n", Book2.author)
   fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
   fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
}
```

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.

Basic Syntax:

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

var *a* = make(map[KeyType]ValueType)
b := make(map[KeyType]ValueType)

Creating an empty Map

var a map[KeyType] ValueType

Accessing elements of a Map

value = map_name[key]

Removing element from Map

delete(map_name, key)

Check if element exists in Map

val, exists := map_name[key]

if exists true value is present in map

```
package main
import ("fmt")

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

  fmt.Println(a)
  fmt.Println(b)

var c = make(map[string]string) // The map is empty now
  c["brand"] = "Renault"
  c["model"] = "Kwid"
  c["year"] = "1964"

  d := make(map[string]int)
  d["Oslo"] = 1
  d["Bergen"] = 2
  d["Trondheim"] = 3
  d["Stavanger"] = 4

  fmt.Println(c)
  fmt.Println(d)

  var e =  make(map[string]string)
 e["brand"] = "Renault"
  e["model"] = "Kwid"
  e["year"] = "1964"

fmt.Println(“Value of Brand in Map a: ”, a[“brand”])
a[“brand”] = “Kia”
fmt.Println(“Value of Brand in Map a: ”, a[“brand”])


  val1, ok1 := a["brand"] // Checking for existing key and its value
  val2, ok2 := a["color"] // Checking for non-existing key and its value
  val3, ok3 := a["day"]   // Checking for existing key and its value
  _, ok4 := a["model"]    // Only checking for existing key and not its value

  fmt.Println(val1, ok1)
  fmt.Println(val2, ok2)
  fmt.Println(val3, ok3)
  fmt.Println(ok4)
}
```

Init

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.

This function is declared implicitly, so you cannot reference it from anywhere and you are allowed to create multiple init() functions in the same program and they execute in the order they are created.

You are allowed to create init() function anywhere in the program and they are called in lexical file name order (Alphabetical Order).

init() function is executed before the main() function call, so it does not depend on the main() function.

The main purpose of the init() function is to initialize the global variables that cannot be initialized in the global context.

Go Routine

The parts of an application that run concurrently are called goroutines. Goroutines and channels are used for structuring concurrent programs.

A process is an independently executing entity running in a machine which runs in its own address space in memory. A process has threads which are simultaneously executing entities. Threads share the same address space of the process.

Goroutines are lightweight, much lighter than a thread. Goroutines run in the same address space, so access to shared memory must be synchronized.

You can goroutines by simply using go keyword before running the required function in a goroutine.

Basic Syntax:

```
func name(){
// statements

}

// using go keyword as the

go name()
```

```go
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("to ACTS")
}

func main() {

    fmt.Println("Welcome!! to Main function")

    // Creating Anonymous Goroutine
    go func() {

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

    time.Sleep(1 * time.Second)
    fmt.Println("GoodBye!! to Main function")
}
```

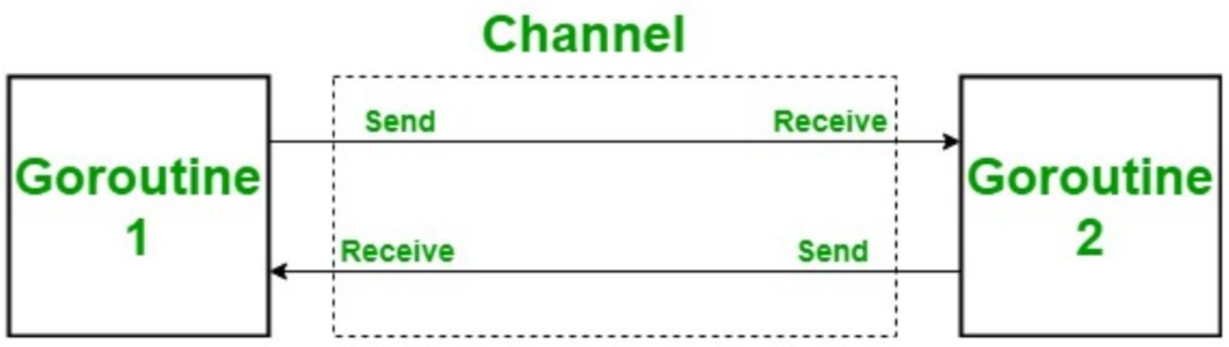
```
package main
import (
    "fmt"
    "time"
)
func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}
func main() {

    f("direct")

    go f("goroutine")

    go func(msg string) {
        fmt.Println(msg)
    }("going")
.
    time.Sleep(time.Second)
    fmt.Println("done")
}
```

Channels

A channel is a medium through which a goroutine communicates with another goroutine and this communication is lock-free.

By default channel is bidirectional, means the goroutines can send or receive data through the same channel.

It can only transfer data of the same type, different types of data are not allowed to transport from the same channel.



Basic Syntax:

var Channel_name chan Type

channel_name:= make(chan Type)

Send Opeartion:

Mychannel <- element

Receive Operation:

element := <-Mychannel

```
package main

import "fmt"

func myfunc(ch chan int) {

    fmt.Println(234 + <-ch)
}
func myfunc1(mychnl chan string) {

    for v := 0; v < 4; v++ {
        mychnl <- "CDAC ACTS"
    }
    close(mychnl)
}
func main() {
    fmt.Println("start Main method")
    // Creating a channel
    ch := make(chan int)

    go myfunc(ch)
    ch <- 23
// Creating a channel
    c := make(chan string)

    // calling Goroutine
    go myfunc1(c)

    // When the value of ok is
    // set to true means the
    // channel is open and it
    // can send or receive data
    // When the value of ok is set to
    // false means the channel is closed
    for {
        res, ok := <-c
        if ok == false {
            fmt.Println("Channel Close ", ok)
            break
        }
        fmt.Println("Channel Open ", res, ok)
    }
    fmt.Println("End Main method")
}
```

```
package main

import "fmt"

// Main function
func main() {

    // Creating a channel
    // Using make() function
    mychnl := make(chan string, 4)
    mychnl <- "CDAC"
    mychnl <- "ACTS"
    mychnl <- "PGDHPCAP"


    // Finding the length and capacity of the channel

    fmt.Println("Length of the channel is: ", len(mychnl))
    fmt.Println("Capacity of the channel is: ", cap(mychnl))
    fmt.Printf("Channel Type: %T\n", mychnl)
    fmt.Printf("Channel Value: %v", mychnl)

}
```

Select

The select statement is just like switch, but in the select statement, case statement refers to communication, i.e. sent or receive operation on the channel.

Select statement waits until the communication(send or receive operation) is prepared for some cases to begin.

```
package main

import "fmt"

// main function
func main() {

 // creating channel
 mychannel:= make(chan int)
select{
 case <- mychannel:

 default:fmt.Println("Not found")
}
}
```

```
// Go program to illustrate the
// concept of select statement
package main

import "fmt"


 // function 1
 func portal1(channel1 chan string){
  for i := 0; i <= 3; i++{
   channel1 <- "Welcome to channel 1"
  }

 }

 // function 2
 func portal2(channel2 chan string){
  channel2 <- "Welcome to channel 2"
 }

// main function
func main() {

 // Creating channels
R1:= make(chan string)
R2:= make(chan string)

// calling function 1 and
// function 2 in goroutine
go portal1(R1)
go portal2(R2)

// the choice of selection
// of case is random
select{
 case op1:= <- R1:
 fmt.Println(op1)
 case op2:= <- R2:
 fmt.Println(op2)
}
}
```