

Golang Tutorial: Learn Go Programming Language for Beginners

What is Go?

Go (also known as Golang) is an open source programming language developed by Google. It is a statically-typed compiled language. Go supports concurrent programming, i.e. it allows running multiple processes simultaneously. This is achieved using channels, goroutines, etc. Go Language has garbage collection which itself does the memory management and allows the deferred execution of functions.

We will learn all the basics of Golang in this Learn Go Language Tutorial.

What You Will Learn: [show]

How to Download and install GO

Step 1) Go to <https://golang.org/dl/>. Download the binary for your OS.



Downloads

After downloading a binary release suitable for your system, please follow the [installation instructions](#).

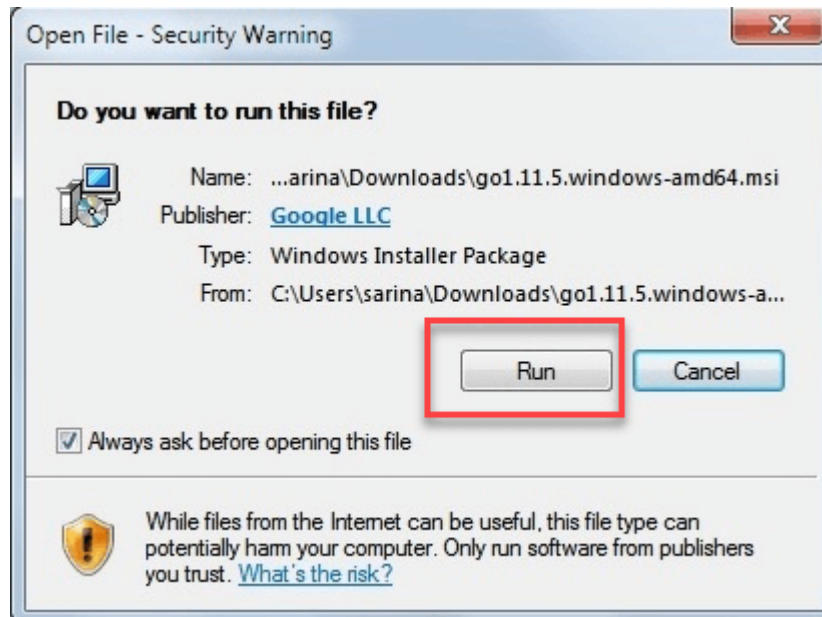
If you are building from source, follow the [source installation instructions](#).

See the [release history](#) for more information about Go releases.

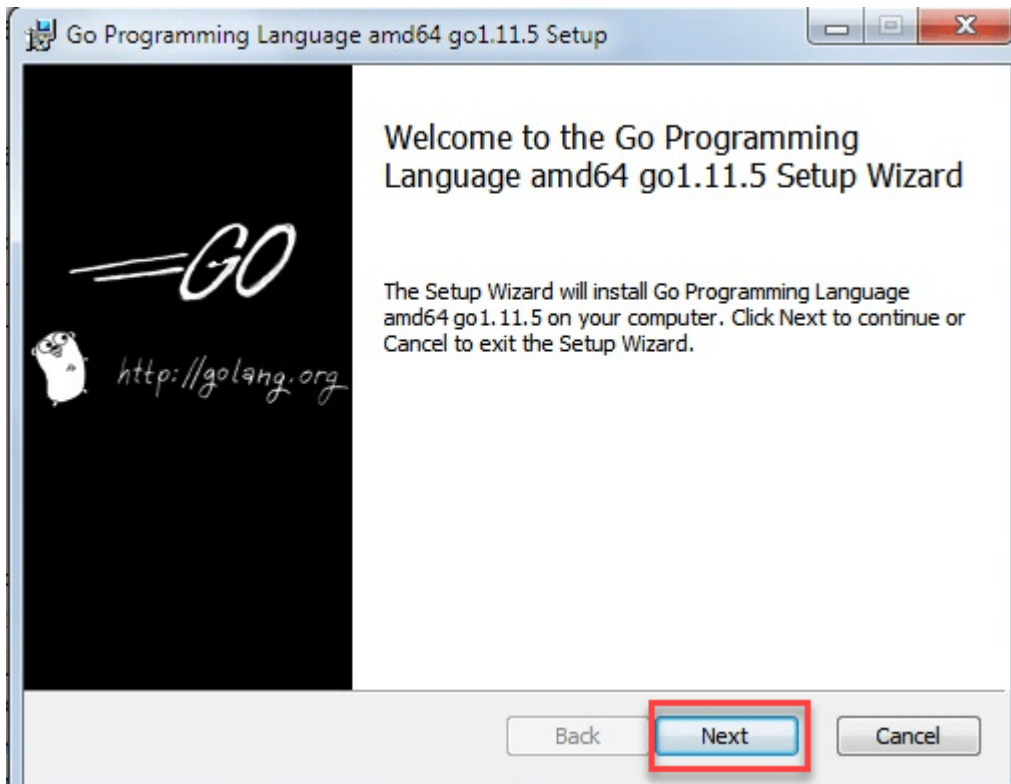
Featured downloads

| | | |
|--|---|--|
| Microsoft Windows <i>Windows 7 or later, Intel 64-bit processor</i> go1.11.5.windows-amd64.msi (111MB) | Apple macOS <i>macOS 10.10 or later, Intel 64-bit processor</i> go1.11.5.darwin-amd64.pkg (114MB) | Linux <i>Linux 2.6.23 or later, Intel 64-bit processor</i> go1.11.5.linux-amd64.tar.gz (134MB) |
|--|---|--|

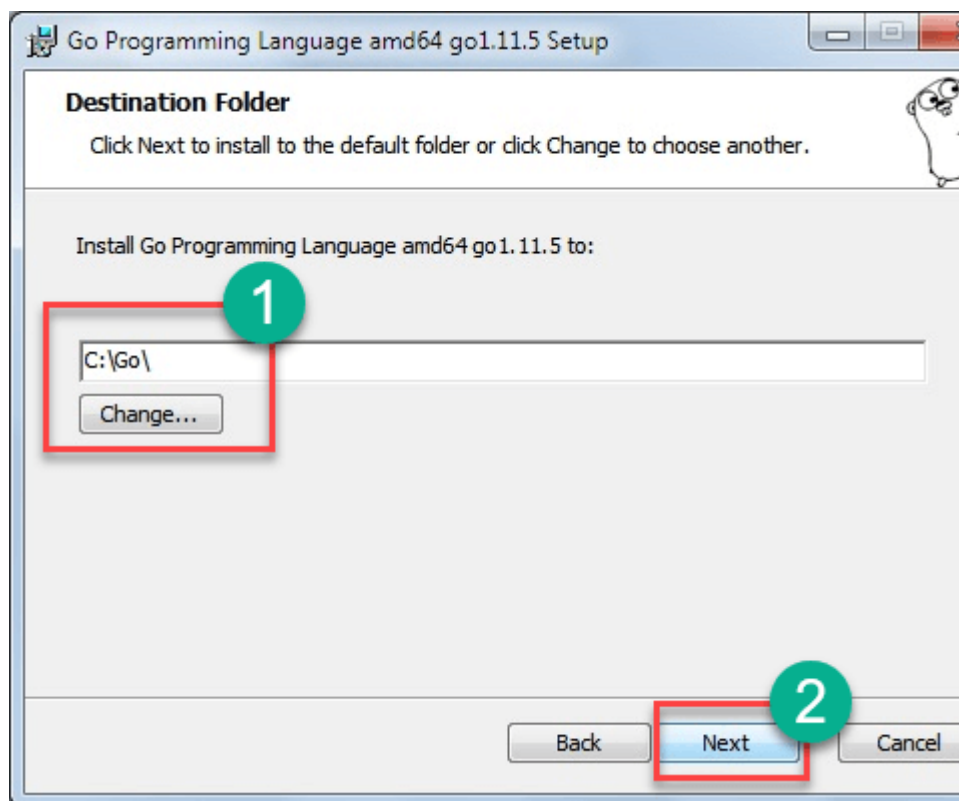
Step 2) Double click on the installer and click Run.



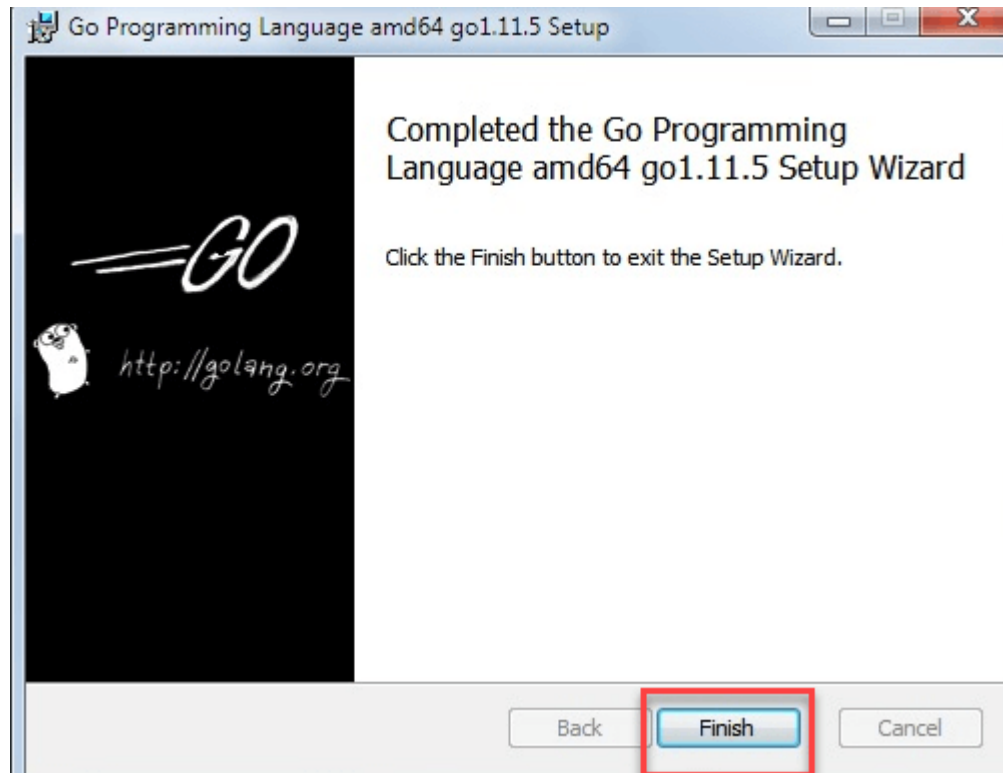
Step 3) Click Next



Step 4) Select the installation folder and click Next.



Step 5) Click Finish once the installation is complete.



Step 6) Once the installation is complete you can verify it by opening the terminal and typing

```
go version
```

This will display the version of go installed

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.

C:\Users\>go version
go version go1.11.5 windows/amd64
```

Your First Go program - Go Hello World!

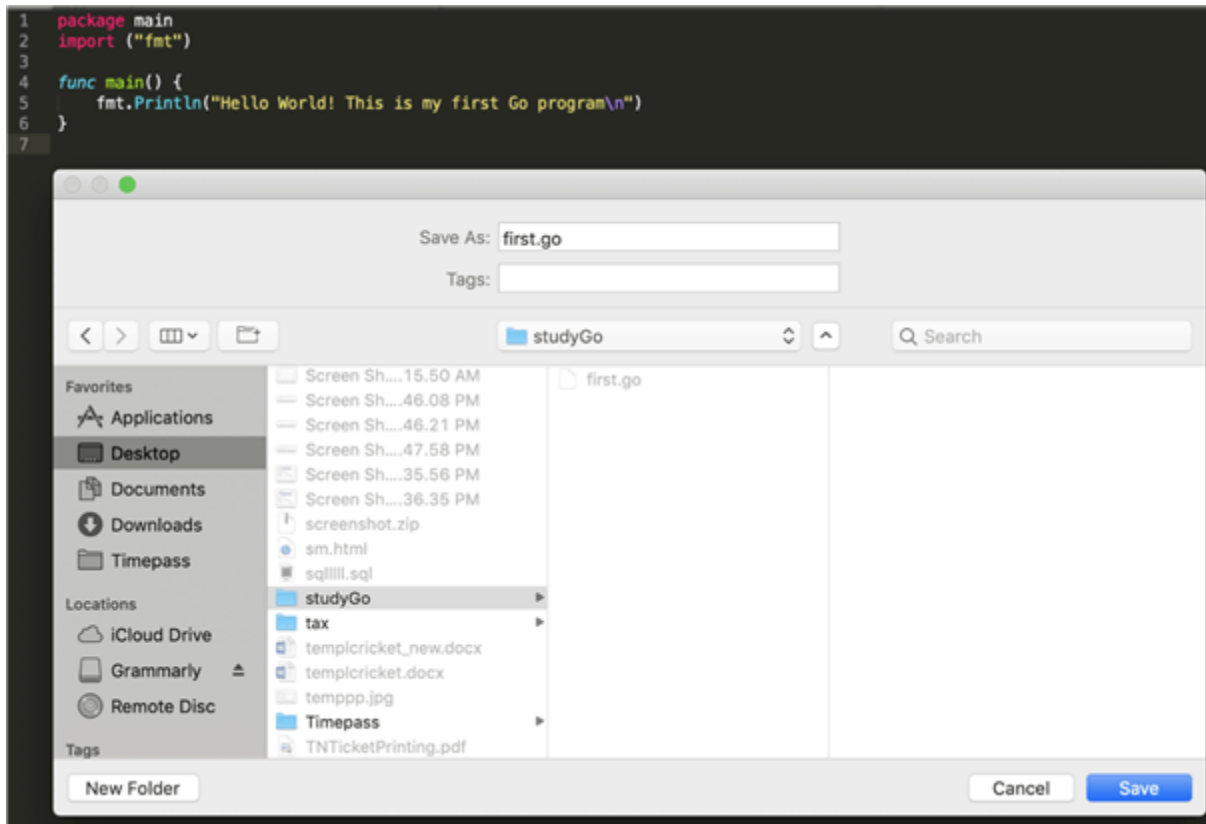
Create a folder called studyGo. In this Go language tutorial, we will create our go programs inside this folder. Go files are created with the extension **.go**. You can run Go programs using the syntax

```
go run <filename>
```

Create a file called first.go and add the below code into it and save

```
package main
import ("fmt")

func main() {
    fmt.Println("Hello World! This is my first Go program\n")
}
```

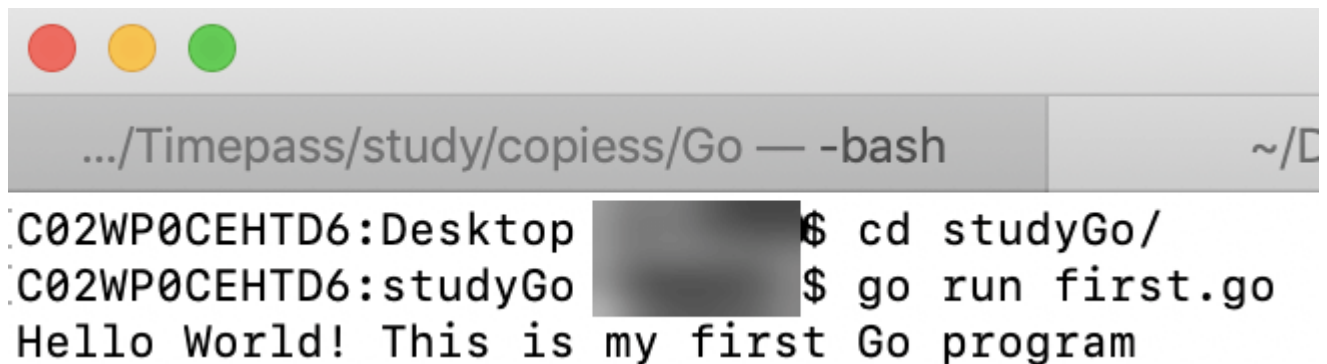


Navigate to this folder in your terminal. Run the program using the command

`go run first.go`

You can see the output printing

```
Hello World! This is my first Go program
```

A terminal window with a title bar containing three colored circles (red, yellow, green). The title bar text is ".../Timepass/study/copiess/Go — -bash" and the user's home directory is "~/". The terminal shows the following commands and output:

```
C02WP0CEHTD6:Desktop $ cd studyGo/  
C02WP0CEHTD6:studyGo $ go run first.go  
Hello World! This is my first Go program
```

Now let's discuss the above program.

package main - Every Go Language program should start with a package name. Go allows us to use packages in another go programs and hence supports code reusability. Execution of a Go program begins with the code inside the package named main.

import fmt - imports the package fmt. This package implements the I/O functions.

func main() - This is the function from which program execution begins. The main function should always be placed in the main package. Under the main(), You can write the code inside { }.

fmt.Println - This will print the text on the screen by the Println function of fmt.

Note: In the below sections of this Go tutorial, when You mention execute/run the code, it means to save the code in a file with .go extension and run it using the syntax

```
go run <filename>
```

Data Types

Types(data types) represent the type of the value stored in a variable, type of the value a function returns, etc.

There are three basic types in Go Language

Numeric types - Represent numeric values which includes integer, floating point, and complex values. Various numeric types are:

int8 - 8 bit signed integers.

int16 - 16 bit signed integers.

int32 - 32 bit signed integers.

int64 - 64 bit signed integers.

uint8 - 8 bit unsigned integers.

uint16 - 16 bit unsigned integers.

uint32 - 32 bit unsigned integers.

uint64 - 64 bit unsigned integers.

float32 - 32 bit floating point numbers.

float64 - 64 bit floating point numbers.

complex64 – has float32 real and imaginary parts.

complex128 - has float32 real and imaginary parts.

String types - Represents a sequence of bytes(characters). You can do various operations on strings like string concatenation, extracting substring, etc

Boolean types - Represents 2 values, either true or false.

Golang Interface

Golang Interface is a collection of method signatures used by a Type to implement the behavior of objects. The main goal of Golang interface is to provide method signatures with names, arguments, and return types. It is up to a Type to declare and implement the method. An interface in Golang can be declared using the keyword “interface.”

Variables

Variables point to a memory location which stores some kind of value. The type parameter(in the below syntax) represents the type of value that can be stored in the memory location.

Variable can be declared using the syntax

```
var <variable_name> <type>
```

Once You declare a variable of a type You can assign the variable to any value of that type.

You can also give an initial value to a variable during the declaration itself using

```
var <variable_name> <type> = <value>
```

If You declare the variable with an initial value, Go can infer the type of the variable from the type of value assigned. So You can omit the type during the declaration using the syntax

```
var <variable_name> = <value>
```

Also, You can declare multiple variables with the syntax

```
var <variable_name1>, <variable_name2> = <value1>, <value2>
```

The below program in this Go tutorial has some Golang examples of variable declarations

```
package main
import "fmt"

func main() {
    //declaring a integer variable x
    var x int
    x=3 //assigning x the value 3
    fmt.Println("x:", x) //prints 3

    //declaring a integer variable y with value 20 in a single statement and
    prints it
    var y int=20
    fmt.Println("y:", y)

    //declaring a variable z with value 50 and prints it
    //Here type int is not explicitly mentioned
    var z=50
    fmt.Println("z:", z)

    //Multiple variables are assigned in single line- i with an integer and j
    with a string
    var i, j = 100,"hello"
    fmt.Println("i and j:", i,j)
}
```

The output will be

```
x: 3
y: 20
z: 50
i and j: 100 hello
```

Go Language also provides an easy way of declaring the variables with value by omitting the var keyword using

```
<variable_name> := <value>
```

Note that You used := instead of =. You cannot use := just to assign a value to a variable which is already declared. := is used to declare and assign value.

Create a file called assign.go with the following code

```
package main
import ("fmt")

func main() {
    a := 20
    fmt.Println(a)

    //gives error since a is already declared
    a := 30
    fmt.Println(a)
}
```

Execute go run assign.go to see the result as

```
./assign.go:7:4: no new variables on left side of :=
```

Variables declared without an initial value will have of 0 for numeric types, false for Boolean and empty string for strings

Constants

Constant variables are those variables whose value cannot be changed once assigned. A constant in Go programming language is declared by using the keyword "const"

Create a file called constant.go and with the following code

```
package main
import ("fmt")

func main() {
    const b =10
    fmt.Println(b)
    b = 30
    fmt.Println(b)
}
```

Execute go run constant.go to see the result as

```
.constant.go:7:4: cannot assign to b
```

For Loop Examples

Loops are used to execute a block of statements repeatedly based on a condition. Most of the programming languages provide 3 types of loops - for, while, do while. **But Go programming language supports only for loop.**

The syntax of a Golang for loop is

```
for initialisation_expression; evaluation_expression; iteration_expression{
    // one or more statement
}
```

The `initialisation_expression` is executed first(and only once) in Golang for loop.

Then the `evaluation_expression` is evaluated and if it's true the code inside the block is executed.

The `iteration_expression` is executed, and the `evaluation_expression` is evaluated again. If it's true the statement block gets executed again. This will continue until the `evaluation_expression` becomes false.

Copy the below program into a file and execute it to see the Golang for loop printing numbers from 1 to 5

```
package main
import "fmt"

func main() {
    var i int
    for i = 1; i <= 5; i++ {
        fmt.Println(i)
    }
}
```

Output is

```
1
2
3
4
5
```

If else

If else is a conditional statement. The syntax is

```
if condition{
    // statements_1
}else{
    // statements_2
}
```

Here the condition is evaluated and if it's true `statements_1` will be executed else `statements_2` will be executed.

You can use if statement without else also. You also can have chained if else statements. The below programs will explain more about if else.

Execute the below program. It checks if a number, x, is less than 10. If so, it will print "x is less than 10"

```
package main
import "fmt"

func main() {
    var x = 50
    if x < 10 {
        //Executes if x < 10
        fmt.Println("x is less than 10")
    }
}
```

Here since the value of x is greater than 10, the statement inside if block condition will not executed.

Now see the below program. In this Go programming language tutorial, we have an else block which will get executed on the failure of if evaluation.

```
package main
import "fmt"

func main() {
    var x = 50
    if x < 10 {
        //Executes if x is less than 10
        fmt.Println("x is less than 10")
    } else {
        //Executes if x >= 10
        fmt.Println("x is greater than or equals 10")
    }
}
```

This program will give you output

```
x is greater than or equals 10
```

Now in this Go tutorial, we will see a program with multiple if else blocks(chained if else). Execute the below Go example. It checks whether a number is less than 10 or is between 10-90 or greater than 90.

```
package main
import "fmt"

func main() {
    var x = 100
    if x < 10 {
        //Executes if x is less than 10
        fmt.Println("x is less than 10")
    } else if x >= 10 && x <= 90 {
        //Executes if x >= 10 and x<=90
        fmt.Println("x is between 10 and 90")
    }
}
```

```

    } else {
        //Executes if both above cases fail i.e x>90
        fmt.Println("x is greater than 90")
    }
}

```

Here first the if condition checks whether x is less than 10 and it's not. So it checks the next condition(else if) whether it's between 10 and 90 which is also false. So it then executes the block under the else section which gives the output

```
x is greater than 90
```

Switch

Switch is another conditional statement. Switch statements evaluate an expression and the result is compared against a set of available values(cases). Once a match is found the statements associated with that match(case) is executed. If no match is found nothing will be executed. You can also add a default case to switch which will be executed if no other matches are found. The syntax of the switch is

```

switch expression {
    case value_1:
        statements_1
    case value_2:
        statements_2
    case value_n:
        statements_n
    default:
        statements_default
}

```

Here the value of the expression is compared against the values in each case. Once a match is found the statements associated with that case is executed. If no match is found the statements under the default section is executed.

Execute the below program

```

package main
import "fmt"

func main() {
    a,b := 2,1
    switch a+b {
    case 1:
        fmt.Println("Sum is 1")
    case 2:
        fmt.Println("Sum is 2")
    case 3:
        fmt.Println("Sum is 3")
    default:
        fmt.Println("Printing default")
    }
}

```

```
}  
}
```

You will get the output as

```
Sum is 3
```

Change the value of a and b to 3 and the result will be

```
Printing default
```

You can also have multiple values in a case by separating them with a comma.

Arrays

Array represents a fixed size, named sequence of elements of the same type. You cannot have an array which contains both integer and characters in it. You cannot change the size of an array once You define the size.

The syntax for declaring an array is

```
var arrayname [size] type
```

Each array element can be assigned value using the syntax

```
arrayname [index] = value
```

Array index starts from **0 to size-1**.

You can assign values to array elements during declaration using the syntax

```
arrayname := [size] type {value_0,value_1,...,value_size-1}
```

You can also ignore the size parameter while declaring the array with values by replacing size with ... and the compiler will find the length from the number of values. Syntax is

```
arrayname := [...] type {value_0,value_1,...,value_size-1}
```

You can find the length of the array by using the syntax

```
len(arrayname)
```

Execute the below Go example to understand the array

```
package main  
import "fmt"  
  
func main() {
```

```

    var numbers [3] string //Declaring a string array of size 3 and adding
elements
    numbers[0] = "One"
    numbers[1] = "Two"
    numbers[2] = "Three"
    fmt.Println(numbers[1]) //prints Two
    fmt.Println(len(numbers)) //prints 3
    fmt.Println(numbers) // prints [One Two Three]

    directions := [...] int {1,2,3,4,5} // creating an integer array and the
size of the array is defined by the number of elements
    fmt.Println(directions) //prints [1 2 3 4 5]
    fmt.Println(len(directions)) //prints 5

    //Executing the below commented statement prints invalid array index 5
(out of bounds for 5-element array)
    //fmt.Println(directions[5])
}

```

Output

```

Two
3
[One Two Three]
[1 2 3 4 5]
5

```

Golang Slice and Append Function

A slice is a portion or segment of an array. Or it is a view or partial view of an underlying array to which it points. You can access the elements of a slice using the slice name and index number just as you do in an array. You cannot change the length of an array, but you can change the size of a slice.

Contents of a slice are actually the pointers to the elements of an array. It means **if you change any element in a slice, the underlying array contents also will be affected.**

The syntax for creating a slice is

```
var slice_name [] type = array_name[start:end]
```

This will create a slice named slice_name from an array named array_name with the elements at the index start to end-1.

Now in this Golang tutorial, we will execute the below program. The program will create a slice from the array and print it. Also, you can see that modifying the contents in the slice will modify the actual array.

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

```

func main() {
    // declaring array
    a := [5] string {"one", "two", "three", "four", "five"}
    fmt.Println("Array after creation:",a)

    var b [] string = a[1:4] //created a slice named b
    fmt.Println("Slice after creation:",b)

    b[0]="changed" // changed the slice data
    fmt.Println("Slice after modifying:",b)
    fmt.Println("Array after slice modification:",a)
}

```

This will print result as

```

Array after creation: [one two three four five]
Slice after creation: [two three four]
Slice after modifying: [changed three four]
Array after slice modification: [one changed three four five]

```

There are certain functions like Golang len, Golang append which you can apply on slices

len(slice_name) - returns the length of the slice

append(slice_name, value_1, value_2) - Golang append is used to append value_1 and value_2 to an existing slice.

append(slice_name1,slice_name2...) – appends slice_name2 to slice_name1

Execute the following program.

```

package main
import "fmt"

func main() {
    a := [5] string {"1","2","3","4","5"}
    slice_a := a[1:3]
    b := [5] string {"one","two","three","four","five"}
    slice_b := b[1:3]

    fmt.Println("Slice_a:", slice_a)
    fmt.Println("Slice_b:", slice_b)
    fmt.Println("Length of slice_a:", len(slice_a))
    fmt.Println("Length of slice_b:", len(slice_b))

    slice_a = append(slice_a,slice_b...) // appending slice
    fmt.Println("New Slice_a after appending slice_b :", slice_a)

    slice_a = append(slice_a,"text1") // appending value
    fmt.Println("New Slice_a after appending text1 :", slice_a)
}

```

The output will be

```

Slice_a: [2 3]
Slice_b: [two three]
Length of slice_a: 2
Length of slice_b: 2
New Slice_a after appending slice_b : [2 3 two three]
New Slice_a after appending text1 : [2 3 two three text1]

```

The program first creates 2 slices and printed its length. Then it appended one slice to other and then appended a string to the resulting slice.

Functions

A function represents a block of statements which performs a specific task. A function declaration tells us function name, return type and input parameters. Function definition represents the code contained in the function. The syntax for declaring the function is

```

func function_name(parameter_1 type, parameter_n type) return_type {
//statements
}

```

The parameters and return types are optional. Also, you can return multiple values from a function.

Now in this Golang tutorial, let's run the following Golang example. Here function named calc will accept 2 numbers and performs the addition and subtraction and returns both values.

```

package main
import "fmt"

//calc is the function name which accepts two integers num1 and num2
//(int, int) says that the function returns two values, both of integer type.
func calc(num1 int, num2 int)(int, int) {
    sum := num1 + num2
    diff := num1 - num2
    return sum, diff
}

func main() {
    x,y := 15,10

    //calls the function calc with x and y and gets sum, diff as output
    sum, diff := calc(x,y)
    fmt.Println("Sum",sum)
    fmt.Println("Diff",diff)
}

```

The output will be

```

Sum 25
Diff 5

```


Packages

Packages are used to organize the code. In a big project, it is not feasible to write code in a single file. Go programming language allow us to organize the code under different packages. This increases code readability and reusability. An executable Go program should contain a package named main and the program execution starts from the function named main. You can import other packages in our program using the syntax

```
import package_name
```

We will see and discuss in this Golang tutorial, how to create and use packages in the following Golang example.

Step 1) Create a file called package_example.go and add the below code

```
package main
import "fmt"
//the package to be created
import "calculation"

func main() {
    x,y := 15,10
    //the package will have function Do_add()
    sum := calculation.Do_add(x,y)
    fmt.Println("Sum", sum)
}
```

In the above program fmt is a package which Go programming language provides us mainly for I/O purposes. Also, you can see a package named calculation. Inside the main() you can see a step `sum := calculation.Do_add(x,y)`. It means you are invoking the function `Do_add` from package calculation.

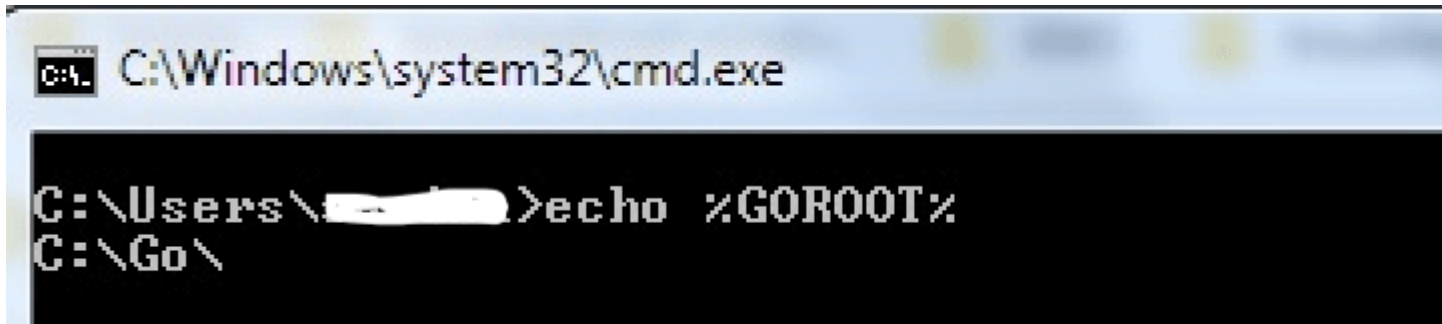
Step 2) First, you should create the package calculation inside a folder with the same name under src folder of the go. The installed path of go can be found from the PATH variable.

For mac, find the path by executing `echo $PATH`

```
C02WP0CEHTD6:calculation i330746$ echo $PATH
/usr/local/opt/curl/bin:/usr/local/opt/openssl/bin:
n:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/go/bin
```

So the path is `/usr/local/go`

For windows, find the path by executing `echo %GOROOT%`



```
C:\Windows\system32\cmd.exe

C:\Users\>echo %GOROOT%
C:\Go\
```

Here the path is C:\Go\

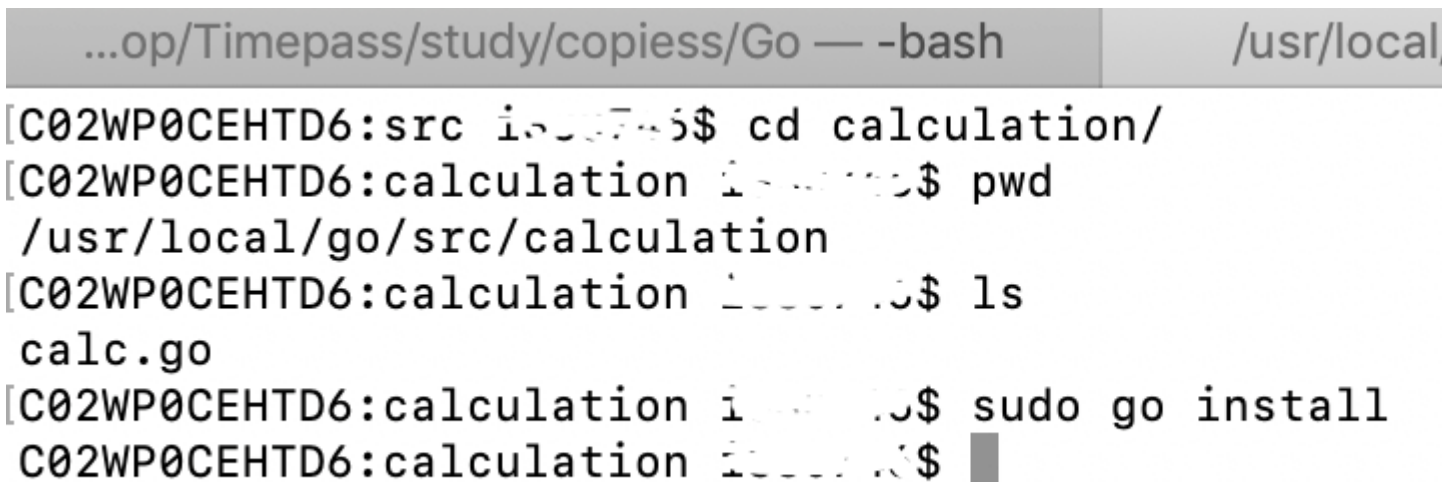
Step 3) Navigate to to the src folder(/usr/local/go/src for mac and C:\Go\src for windows). Now from the code, the package name is calculation. Go requires the package should be placed in a directory of the same name under src directory. Create a directory named calculation in src folder.

Step 4) Create a file called calc.go (You can give any name, but the package name in the code matters. Here it should be calculation) inside calculation directory and add the below code

```
package calculation

func Do_add(num1 int, num2 int)(int) {
    sum := num1 + num2
    return sum
}
```

Step 5) Run the command go install from the calculation directory which will compile the calc.go.



```
...op/Timepass/study/copiess/Go — -bash /usr/local/

[C02WP0CEHTD6:src i-03745$ cd calculation/
[C02WP0CEHTD6:calculation i-03745$ pwd
/usr/local/go/src/calculation
[C02WP0CEHTD6:calculation i-03745$ ls
calc.go
[C02WP0CEHTD6:calculation i-03745$ sudo go install
C02WP0CEHTD6:calculation i-03745$
```

Step 6) Now go back to package_example.go and run go run package_example.go. The output will be Sum 25.

Note that the name of the function `Do_add` starts with a capital letter. This is because in Go if the function name starts with a capital letter it means other programs can see(access) it else other programs cannot access it. If the function name was `do_add` , then You would have got the error

cannot refer to unexported name `calculation.calc..`

Defer and stacking defers

Defer statements are used to defer the execution of a function call until the function that contains the defer statement completes execution.

Lets learn this with an example:

```
package main
import "fmt"

func sample() {
    fmt.Println("Inside the sample()")
}
func main() {
    //sample() will be invoked only after executing the statements of main()
    defer sample()
    fmt.Println("Inside the main()")
}
```

The output will be

```
Inside the main()
Inside the sample()
```

Here execution of `sample()` is deferred until the execution of the enclosing function(`main()`) completes.

Stacking defer is using multiple defer statements. Suppose you have multiple defer statements inside a function. Go places all the deferred function calls in a stack, and once the enclosing function returns, the stacked functions are executed in the **Last In First Out(LIFO) order**. You can see this in the below example.

Execute the below code

```
package main
import "fmt"

func display(a int) {
    fmt.Println(a)
}
func main() {
    defer display(1)
    defer display(2)
    defer display(3)
```

```
    fmt.Println(4)
}
```

The output will be

```
4
3
2
1
```

Here the code inside the main() executes first, and then the deferred function calls are executed in the reverse order, i.e. 4, 3, 2, 1.

Pointers

Before explaining pointers let's will first discuss '&' operator. The '&' operator is used to get the address of a variable. It means '&a' will print the memory address of variable a.

In this Golang tutorial, we will execute the below program to display the value of a variable and the address of that variable

```
package main
import "fmt"

func main() {
    a := 20
    fmt.Println("Address:", &a)
    fmt.Println("Value:", a)
}
```

The result will be

```
Address: 0xc000078008
Value: 20
```

A pointer variable stores the memory address of another variable. You can define a pointer using the syntax

```
var variable_name *type
```

The asterisk(*) represents the variable is a pointer. You will understand more by executing the below program

```
package main
import "fmt"

func main() {
    //Create an integer variable a with value 20
    a := 20
```

```

//Create a pointer variable b and assigned the address of a
var b *int = &a

//print address of a(&a) and value of a
fmt.Println("Address of a:", &a)
fmt.Println("Value of a:", a)

//print b which contains the memory address of a i.e. &a
fmt.Println("Address of pointer b:", b)

// *b prints the value in memory address which b contains i.e. the
value of a
fmt.Println("Value of pointer b", *b)

//increment the value of variable a using the variable b
*b = *b+1

//prints the new value using a and *b
fmt.Println("Value of pointer b", *b)
fmt.Println("Value of a:", a)

```

The output will be

```

Address of a: 0x416020
Value of a: 20
Address of pointer b: 0x416020
Value of pointer b 20
Value of pointer b 21
Value of a: 21

```

Structures

A Structure is a user defined datatype which itself contains one more element of the same or different type.

Using a structure is a 2 step process.

First, create(declare) a structure type

Second, create variables of that type to store values.

Structures are mainly used when you want to store related data together.

Consider a piece of employee information which has name, age, and address. You can handle this in 2 ways

Create 3 arrays - one array stores the names of employees, one stores age and the third one stores age.

Declare a structure type with 3 fields- name, address, and age. Create an array of that structure type where each element is a structure object having name, address, and age.

The first approach is not efficient. In these kinds of scenarios, structures are more convenient.

The syntax for declaring a structure is

```
type structname struct {
    variable_1 variable_1_type
    variable_2 variable_2_type
    variable_n variable_n_type
}
```

An example of a structure declaration is

```
type emp struct {
    name string
    address string
    age int
}
```

Here a new user defined type named emp is created. Now, you can create variables of the type emp using the syntax

```
var variable_name struct_name
```

An example is

```
var empdata1 emp
```

You can set values for the empdata1 as

```
empdata1.name = "John"
empdata1.address = "Street-1, Bangalore"
empdata1.age = 30
```

You can also create a structure variable and assign values by

```
empdata2 := emp{"Raj", "Building-1, Delhi", 25}
```

Here, you need to maintain the order of elements. Raj will be mapped to name, next element to address and the last one to age.

Execute the code below

```
package main
import "fmt"

//declared the structure named emp
type emp struct {
```

```

        name string
        address string
        age int
    }

    //function which accepts variable of emp type and prints name property
    func display(e emp) {
        fmt.Println(e.name)
    }

    func main() {
        // declares a variable, empdata1, of the type emp
        var empdata1 emp
        //assign values to members of empdata1
        empdata1.name = "John"
        empdata1.address = "Street-1, London"
        empdata1.age = 30

        //declares and assign values to variable empdata2 of type emp
        empdata2 := emp{"Raj", "Building-1, Paris", 25}

        //prints the member name of empdata1 and empdata2 using display function
        display(empdata1)
        display(empdata2)
    }

```

Output

```

John
Raj

```

Methods(not functions)

A method is a function with a receiver argument. Architecturally, it's between the func keyword and method name. The syntax of a method is

```

func (variable variabletype) methodName(parameter1 paramether1type) {
}

```

Let's convert the above example program to use methods instead of function.

```

package main
import "fmt"

//declared the structure named emp
type emp struct {
    name string
    address string
    age int
}

//Declaring a function with receiver of the type emp
func(e emp) display() {

```

```

        fmt.Println(e.name)
    }

func main() {
    //declaring a variable of type emp
    var empdata1 emp

    //Assign values to members
    empdata1.name = "John"
    empdata1.address = "Street-1, Lodon"
    empdata1.age = 30

    //declaring a variable of type emp and assign values to members
    empdata2 := emp {
        "Raj", "Building-1, Paris", 25}

    //Invoking the method using the receiver of the type emp
    // syntax is variable.methodname()
    empdata1.display()
    empdata2.display()
}

```

Go is not an object oriented language and it doesn't have the concept of class. Methods give a feel of what you do in object oriented programs where the functions of a class are invoked using the syntax `objectname.functionname()`

Concurrency

Go supports concurrent execution of tasks. It means Go can execute multiple tasks simultaneously. It is different from the concept of parallelism. In parallelism, a task is split into small subtasks and are executed in parallel. But in concurrency, multiple tasks are being executed simultaneously. Concurrency is achieved in Go using Goroutines and Channels.

Goroutines

A goroutine is a function which can run concurrently with other functions. Usually when a function is invoked the control gets transferred into the called function, and once its completed execution control returns to the calling function. The calling function then continues its execution. The calling function waits for the invoked function to complete the execution before it proceeds with the rest of the statements.

But in the case of goroutine, the calling function will not wait for the execution of the invoked function to complete. It will continue to execute with the next statements. You can have multiple goroutines in a program.

Also, the main program will exit once it completes executing its statements and it will not wait for completion of the goroutines invoked.

Goroutine is invoked using keyword `go` followed by a function call.

Example

```
go add(x,y)
```

You will understand goroutines with the below Golang examples. Execute the below program

```
package main
import "fmt"

func display() {
    for i:=0; i<5; i++ {
        fmt.Println("In display")
    }
}

func main() {
    //invoking the goroutine display()
    go display()
    //The main() continues without waiting for display()
    for i:=0; i<5; i++ {
        fmt.Println("In main")
    }
}
```

The output will be

```
In main
In main
In main
In main
In main
```

Here the main program completed execution even before the goroutine started. The display() is a goroutine which is invoked using the syntax

```
go function_name(parameter list)
```

In the above code, the main() doesn't wait for the display() to complete, and the main() completed its execution before the display() executed its code. So the print statement inside display() didn't get printed.

Now we modify the program to print the statements from display() as well. We add a time delay of 2 sec in the for loop of main() and a 1 sec delay in the for loop of the display().

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

func display() {
    for i:=0; i<5; i++ {
        time.Sleep(1 * time.Second)
        fmt.Println("In display")
    }
}
```

```

    }
}

func main() {
    //invoking the goroutine display()
    go display()
    for i:=0; i<5; i++ {
        time.Sleep(2 * time.Second)
        fmt.Println("In main")
    }
}

```

The output will be somewhat similar to

```

In display
In main
In display
In display
In main
In display
In display
In main
In main
In main

```

Here You can see both loops are being executed in an overlapping fashion because of the concurrent execution.

Channels

Channels are a way for functions to communicate with each other. It can be thought as a medium to where one routine places data and is accessed by another routine in Golang server.

A channel can be declared with the syntax

```
channel_variable := make(chan datatype)
```

Example:

```
ch := make(chan int)
```

You can send data to a channel using the syntax

```
channel_variable <- variable_name
```

Example

```
ch <- x
```

You can receive data from a channel using the syntax

```
variable_name := <- channel_variable
```

Example

```
y := <- ch
```

In the above Go language examples of goroutine, you have seen the main program doesn't wait for the goroutine. But that is not the case when channels are involved. Suppose if a goroutine pushes data to channel, the main() will wait on the statement receiving channel data until it gets the data.

You will see this in below Go language examples. First, write a normal goroutine and see the behaviour. Then modify the program to use channels and see the behaviour.

Execute the below program

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

func display() {
    time.Sleep(5 * time.Second)
    fmt.Println("Inside display()")
}

func main() {
    go display()
    fmt.Println("Inside main()")
}
```

The output will be

```
Inside main()
```

The main() finished the execution and did exit before the goroutine executes. So the print inside the display() didn't get executed.

Now modify the above program to use channels and see the behaviour.

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

func display(ch chan int) {
    time.Sleep(5 * time.Second)
    fmt.Println("Inside display()")
    ch <- 1234
}

func main() {
    ch := make(chan int)
```

```

        go display(ch)
        x := <-ch
        fmt.Println("Inside main()")
        fmt.Println("Printing x in main() after taking from channel:",x)
    }

```

The output will be

```

Inside display()
Inside main()
Printing x in main() after taking from channel: 1234

```

Here what happens is the main() on reaching `x := <-ch` will wait for data on channel `ch`. The `display()` has a wait of 5 seconds and then push data to the channel `ch`. The main() on receiving the data from the channel gets unblocked and continues its execution.

The sender who pushes data to channel can inform the receivers that no more data will be added to the channel by closing the channel. This is mainly used when you use a loop to push data to a channel. A channel can be closed using

```
close(channel_name)
```

And at the receiver end, it is possible to check whether the channel is closed using an additional variable while fetching data from channel using

```
variable_name, status := <- channel_variable
```

If the status is True it means you received data from the channel. If false, it means you are trying to read from a closed channel

You can also use channels for communication between goroutines. Need to use 2 goroutines – one pushes data to the channel and other receives the data from the channel. See the below program

```

package main
import "fmt"
import "time"

//This subroutine pushes numbers 0 to 9 to the channel and closes the channel
func add_to_channel(ch chan int) {
    fmt.Println("Send data")
    for i:=0; i<10; i++ {
        ch <- i //pushing data to channel
    }
    close(ch) //closing the channel
}

//This subroutine fetches data from the channel and prints it.
func fetch_from_channel(ch chan int) {
    fmt.Println("Read data")
}

```

```

        for {
            //fetch data from channel
            x, flag := <- ch

            //flag is true if data is received from the channel
            //flag is false when the channel is closed
            if flag == true {
                fmt.Println(x)
            }else{
                fmt.Println("Empty channel")
                break
            }
        }
    }

func main() {
    //creating a channel variable to transport integer values
    ch := make(chan int)

    //invoking the subroutines to add and fetch from the channel
    //These routines execute simultaneously
    go add_to_channel(ch)
    go fetch_from_channel(ch)

    //delay is to prevent the exiting of main() before goroutines finish
    time.Sleep(5 * time.Second)
    fmt.Println("Inside main()")
}

```

Here there are 2 subroutines one pushes data to the channel and other prints data to the channel. The function `add_to_channel` adds the numbers from 0 to 9 and closes the channel. Simultaneously the function `fetch_from_channel` waits at

`x, flag := <- ch` and once the data become available, it prints the data. It exits once the flag is false which means the channel is closed.

The wait in the `main()` is given to prevent the exiting of `main()` until the goroutines finish the execution.

Execute the code and see the output as

```

Read data
Send data
0
1
2
3
4
5
6
7
8
9
Empty channel

```

Inside main()

Select

Select can be viewed as a switch statement which works on channels. Here the case statements will be a channel operation. Usually, each case statements will be read attempt from the channel. When any of the cases is ready(the channel is read), then the statement associated with that case is executed. If multiple cases are ready, it will choose a random one. You can have a default case which is executed if none of the cases is ready.

Let's see the below code

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

//push data to channel with a 4 second delay
func data1(ch chan string) {
    time.Sleep(4 * time.Second)
    ch <- "from data1()"
}

//push data to channel with a 2 second delay
func data2(ch chan string) {
    time.Sleep(2 * time.Second)
    ch <- "from data2()"
}

func main() {
    //creating channel variables for transporting string values
    chan1 := make(chan string)
    chan2 := make(chan string)

    //invoking the subroutines with channel variables
    go data1(chan1)
    go data2(chan2)

    //Both case statements wait for data in the chan1 or chan2.
    //chan2 gets data first since the delay is only 2 sec in data2().
    //So the second case will execute and exits the select block
    select {
    case x := <-chan1:
        fmt.Println(x)
    case y := <-chan2:
        fmt.Println(y)
    }
}
```

Executing the above program will give the output:

```
from data2()
```

Here the select statement waits for data to be available in any of the channels. The data2() adds data to the channel after a sleep of 2 seconds which will cause the second case to execute.

Add a default case to the select in the same program and see the output. Here, on reaching select block, if no case is having data ready on the channel, it will execute the default block without waiting for data to be available on any channel.

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

//push data to channel with a 4 second delay
func data1(ch chan string) {
    time.Sleep(4 * time.Second)
    ch <- "from data1()"
}

//push data to channel with a 2 second delay
func data2(ch chan string) {
    time.Sleep(2 * time.Second)
    ch <- "from data2()"
}

func main() {
    //creating channel variables for transporting string values
    chan1 := make(chan string)
    chan2 := make(chan string)

    //invoking the subroutines with channel variables
    go data1(chan1)
    go data2(chan2)

    //Both case statements check for data in chan1 or chan2.
    //But data is not available (both routines have a delay of 2 and 4 sec)
    //So the default block will be executed without waiting for data in
    channels.
    select {
    case x := <-chan1:
        fmt.Println(x)
    case y := <-chan2:
        fmt.Println(y)
    default:
        fmt.Println("Default case executed")
    }
}
```

This program will give the output:

```
Default case executed
```

This is because when the select block reached, no channel had data for reading. So, the default case is executed.

Mutex

Mutex is the short form for mutual exclusion. Mutex is used when you don't want to allow a resource to be accessed by multiple subroutines at the same time. Mutex has 2 methods - Lock and Unlock. Mutex is contained in sync package. So, you have to import the sync package. The statements which have to be mutually exclusively executed can be placed inside mutex.Lock() and mutex.Unlock().

Let's learn mutex with an example which is counting the number of times a loop is executed. In this program we expect routine to run loop 10 times and the count is stored in sum. You call this routine 3 times so the total count should be 30. The count is stored in a global variable count.

First, You run the program without mutex

```
package main
import "fmt"
import "time"
import "strconv"
import "math/rand"
//declare count variable, which is accessed by all the routine instances
var count = 0

//copies count to temp, do some processing(increment) and store back to count
//random delay is added between reading and writing of count variable
func process(n int) {
    //loop incrementing the count by 10
    for i := 0; i < 10; i++ {
        time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
        temp := count
        temp++
        time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
        count = temp
    }
    fmt.Println("Count after i="+strconv.Itoa(n)+" Count:",
    strconv.Itoa(count))
}

func main() {
    //loop calling the process() 3 times
    for i := 1; i < 4; i++ {
        go process(i)
    }

    //delay to wait for the routines to complete
    time.Sleep(25 * time.Second)
    fmt.Println("Final Count:", count)
}
```

See the result

```
Count after i=1 Count: 11
Count after i=3 Count: 12
```



```
Count after i=2 Count: 13
Final Count: 13
```

The result could be different when you execute it but the final result won't be 30.

Here what happens is 3 goroutines are trying to increase the loop count stored in the variable count. Suppose at a moment count is 5 and goroutine1 is going to increment the count to 6. The main steps include

Copy count to temp

Increment temp

Store temp back to count

Suppose soon after performing step 3 by goroutine1; another goroutine might have an old value say 3 does the above steps and store 4 back, which is wrong. This can be prevented by using mutex which causes other routines to wait when one routine is already using the variable.

Now You will run the program with mutex. Here the above mentioned 3 steps are executed in a mutex.

```
package main
import "fmt"
import "time"
import "sync"
import "strconv"
import "math/rand"

//declare a mutex instance
var mu sync.Mutex

//declare count variable, which is accessed by all the routine instances
var count = 0

//copies count to temp, do some processing(increment) and store back to count
//random delay is added between reading and writing of count variable
func process(n int) {
    //loop incrementing the count by 10
    for i := 0; i < 10; i++ {
        time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
        //lock starts here
        mu.Lock()
        temp := count
        temp++
        time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
        count = temp
        //lock ends here
        mu.Unlock()
    }
    fmt.Println("Count after i="+strconv.Itoa(n)+" Count:",
        strconv.Itoa(count))
}
```

```

}

func main() {
    //loop calling the process() 3 times
    for i := 1; i < 4; i++ {
        go process(i)
    }

    //delay to wait for the routines to complete
    time.Sleep(25 * time.Second)
    fmt.Println("Final Count:", count)
}

```

Now the output will be

```

Count after i=3 Count: 21
Count after i=2 Count: 28
Count after i=1 Count: 30
Final Count: 30

```

Here we get the expected result as final output. Because the statements reading, incrementing and writing back of count is executed in a mutex.

Error handling

Errors are abnormal conditions like closing a file which is not opened, open a file which doesn't exist, etc. Functions usually return errors as the last return value.

The below example explains more about the error.

```

package main
import "fmt"
import "os"

//function accepts a filename and tries to open it.
func fileopen(name string) {
    f, er := os.Open(name)

    //er will be nil if the file exists else it returns an error object
    if er != nil {
        fmt.Println(er)
        return
    }else{
        fmt.Println("file opened", f.Name())
    }
}

func main() {
    fileopen("invalid.txt")
}

```

The output will be:

```
open /invalid.txt: no such file or directory
```

Here we tried to open a non-existing file, and it returned the error to `er` variable. If the file is valid, then the error will be null

Custom errors

Using this feature, you can create custom errors. This is done by using `New()` of error package. We will rewrite the above program to make use of custom errors.

Run the below program

```
package main
import "fmt"
import "os"
import "errors"

//function accepts a filename and tries to open it.
func fileopen(name string) (string, error) {
    f, er := os.Open(name)

    //er will be nil if the file exists else it returns an error object
    if er != nil {
        //created a new error object and returns it
        return "", errors.New("Custom error message: File name is wrong")
    }else{
        return f.Name(), nil
    }
}

func main() {
    //receives custom error or nil after trying to open the file
    filename, error := fileopen("invalid.txt")
    if error != nil {
        fmt.Println(error)
    }else{
        fmt.Println("file opened", filename)
    }
}
```

The output will be:

```
Custom error message:File name is wrong
```

Here the `area()` returns the area of a square. If the input is less than 1 then `area()` returns an error message.

Reading files

Files are used to store data. Go allows us to read data from the files

First create a file, data.txt, in your present directory with the below content.

```
Line one  
Line two  
Line three
```

Now run the below program to see it prints the contents of the entire file as output

```
package main  
import "fmt"  
import "io/ioutil"  
  
func main() {  
    data, err := ioutil.ReadFile("data.txt")  
    if err != nil {  
        fmt.Println("File reading error", err)  
        return  
    }  
    fmt.Println("Contents of file:", string(data))  
}
```

Here the data, err := ioutil.ReadFile("data.txt") reads the data and returns a byte sequence. While printing it is converted to string format.

Writing files

You will see this with a program

```
package main  
import "fmt"  
import "os"  
  
func main() {  
    f, err := os.Create("file1.txt")  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    l, err := f.WriteString("Write Line one")  
    if err != nil {  
        fmt.Println(err)  
        f.Close()  
        return  
    }  
    fmt.Println(l, "bytes written")  
    err = f.Close()  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
}
```

Here a file is created, test.txt. If the file already exists then the contents of the file are truncated. Writeline() is used to write the contents to the file. After that, You closed the file using Close().

Cheat Sheet

In this Go tutorial, we covered,

| Topic | Description | Syntax |
|-------------|--|---|
| Basic types | Numeric, string, bool | |
| Variables | Declare and assign values to variables | <pre>var variable_name type var variable_name type = value var variable_name1, variable_name2 = value1, value2 variable_name := value</pre> |
| Constants | Variables whose value cannot be changed once assigned | <pre>const variable = value</pre> |
| For Loop | Execute statements in a loop. | <pre>for initialisation_expression; evaluation_expression; iteration_expression { // one or more statement }</pre> |
| If else | It is a conditional statement | <pre>if condition { // statements_1 } else { // statements_2 }</pre> |
| switch | Conditional statement with multiple cases | <pre>switch expression { case value_1: statements_1 case value_2: statements_2 case value_n: statements_n default: statements_default }</pre> |
| Array | Fixed size named sequence of elements of same type | <pre>arrayname := [size] type {value_0,value_1,...,value_size-1}</pre> |
| Slice | Portion or segment of an array | <pre>var slice_name [] type = array_name[start:end]</pre> |
| Functions | Block of statements which performs a specific task | <pre>func function_name(parameter_1 type, parameter_n type) return_type { //statements }</pre> |
| Packages | Are used to organise the code. Increases code readability and reusability | <pre>import package_nam</pre> |
| Defer | Defers the execution of a function till the containing function finishes execution | <pre>defer function_name(parameter_list)</pre> |
| Pointers | Stores the memory address of another variable. | <pre>var variable_name *type</pre> |
| Structure | User defined datatype which itself contains one more element of the same or different type | <pre>type structname struct { variable_1 variable_1_type variable_2 variable_2_type variable_n variable_n_type }</pre> |
| Methods | A method is a function with a receiver argument | <pre>func (variable variabletype) methodName(parameter_list) { }</pre> |
| Goroutine | A function which can run | <pre>go function_name(parameter_list)</pre> |

| | | |
|------------|---|--|
| | concurrently with other functions. | |
| Channel | Way for functions to communicate with each other. A medium to which one routine places data and is accessed by another routine. | Declare: <code>ch := make(chan int)</code> Send data to channel: <code>channel_variable <- variable_name</code> Receive from channel: <code>variable_name := <- channel_variable</code> |
| Select | Switch statement which works on channels. The case statements will be a channel operation. When any of the channel is ready with data, then the statement associated with that case is executed | <code>select { case x := <-chan1: fmt.Println(x) case y := <-chan2: fmt.Println(y) }</code> |
| Mutex | Mutex is used when you don't want to allow a resource to be accessed by multiple subroutines at the same time. Mutex has 2 methods - Lock and Unlock | <code>mutex.Lock() //statements mutex.Unlock().</code> |
| Read files | Reads the data and returns a byte sequence. | <code>Data, err := ioutil.ReadFile(filename)</code> |
| Write file | Writes data to a file | <code>l, err := f.WriteString(text_to_write)</code> |