Go Language Tutorial



Go tutorial provides basic and advanced concepts of Go programming. Our Go language tutorial is designed for beginners and professionals both.

Go is a programming language which is developed by Google with the vision of fast development and high performance.

Our Go Tutorial includes all topics of Go language such as what is go, how to install go, go if-else, go for, go for-range, go break, go continue, go struct, go interface, go ruin, go map, go string, go array, go http server, go rest api, go mutex etc.

Prerequisite

To learn Go, you must have the basic knowledge of C programming.

# What is Go

Go is a programming language which is developed by Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. Go is a statically-typed language. Go has similar syntax like C. It is developed with the vision of high performance and fast development. Go provides type safety, garbage collection, dynamic-typing capability, many advanced built-in types such as variable length arrays and key-value maps etc.

* Go is modern, fast and comes with a powerful standard library.
* Go has built-in concurrency.
* Go uses interfaces as the building blocks of code reusability.

The basic structure of a Go programs consists of following parts:-

* Package Declaration
* Import Packages
* Variables
* Statements and Expressions
* Functions
* Comments

# Go Installation

You can install Go programming on different operating systems like Windows, Linux, Mac etc. This is a link of binary distribution of the Go programming for FreeBSD (release 8-STABLE and above), Linux, Mac OS X (10.8 and above), and Windows operating systems for the 32-bit (386) and 64-bit (amd64) architectures.

If you do not found configuration of your combination, try installing from source or installing gccgo instead of gc.

### **In Linux**

* We should choose the required archive file for installation. For Example, if we are installing Go version 1.6.1 for 64-bit x86 on Linux, the archive would be go1.2.1.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:

1. 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:

1. export PATH=$PATH:/usr/local/go/bin

### **In Windows**

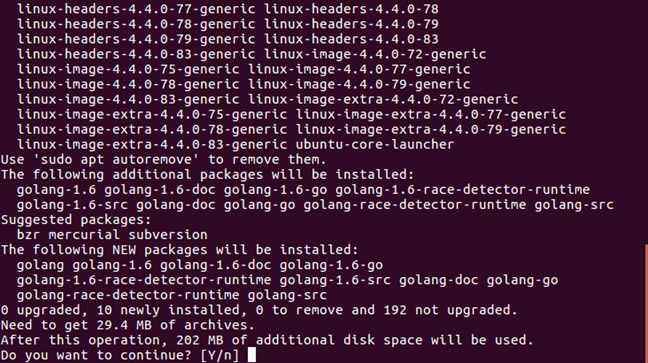
* Choose the required archive file for Windows installation.
* Download zip file and extract it into the directory (Like c:\Go).
* If you have choosed a different directory other than c:\Go, you must set the GOROOT environment variable to your chosen path.
* Add the bin subdirectory of your Go root (for example, c:\Go\bin) to your PATH environment variable.

### **In Mac OS X**

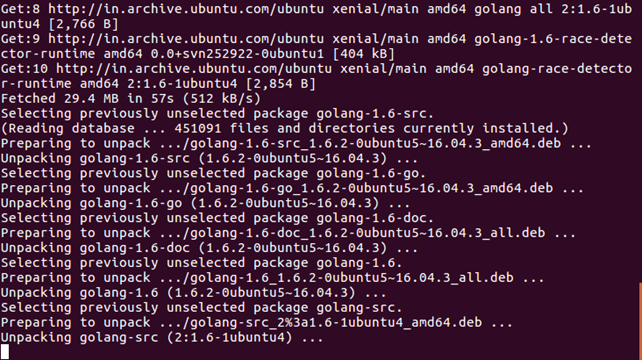
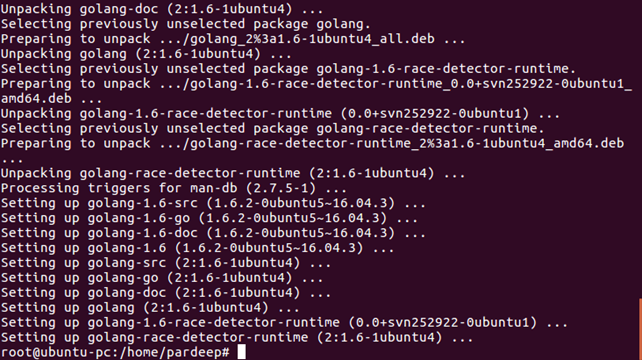
* Choose the required archive file for Mac installation.
* Open the downloaded package file, and follow the prompts to install the Go tools. The package installs the Go distribution to /usr/local/go.
* The package should locate the /usr/local/go/bin directory to your PATH environment variable. You may need to restart the opened terminal sessions to make the change.

Go **Installation in Ubuntu**

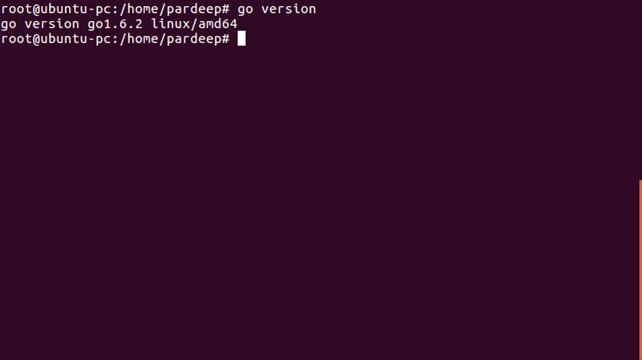
**Step 1:** Run the command **sudo apt-get install golang**

**Step 2:** Press 'y' when you asked

Now, Go installation is done, You can check the version by the command: **go version.**



# Go Hello World Example

1. **package** main
2. **import** "fmt"
3. func main() {
4. fmt.Println("Hello, World")
5. }

Output:

Hello, World

The First line is the package deceleration, here the name of the package is main. Package deceleration is mandatory for all the go programs.

The next line is an import statement, here we are importing "fmt". The compiler will include the files of the package fmt.

The next line is a main() function, all execution begins with the main function.

The next line fmt.Println(...) is a function available in Go. This function will print the message "Hello, World" on the screen.

### **How to run Go program:**

We have saved the program as Hello.go. To run the program, open the command prompt and write:

1. go run Hello.go

Output:

Hello, World

Go Data Types

Variables can be of different types like int, float, struct, slice or it can be of the interface.

The general form for declaring a variable uses the keyword var:

Syntax:-

1. var identifier type

**Example**

1. var a **int**
2. var b bool
3. var str string

when a variable is declared with var it automatically initializes it to the zero-value defined for its type. A type defines the set of values and the set of operations that can take place on those values.

**GO Simple Data Type Example**

1. **package** main
2. **import** "fmt"
3. func main() {
4. var i **int**
5. var f float64
6. var b bool
7. var s string
8. fmt.Printf("%T %T %T %T\n",i,f,b,s) // Prints type of the variable
9. fmt.Printf("%v   %v      %v  %q     \n", i, f, b, s) //prints initial value of the variable
10. }

Output:

int float64 bool string

0 0 false ""

Go Construct and Data Types

The Go source code is stored in .go file. The name of the file consists of lowercase letters. If the file name has several parts, it should be separated by underscore "\_" .

Go file has a name or an identifier which is case sensitive like C.

**For example:** a, ax123, i etc.

The \_ identifier is special. It is called blank identifier. It may be used in variable declarations.

It is like normal identifiers but its value is discarded, so it cannot be used anymore in the code.

It may happen that the variable, type, or function has no name and even enhance flexibility so it is called anonymous.

These are the 25 keywords for Go-code:

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

Programs consist of keywords, constants, variables, operators, types and functions.

The following delimiters are used in constructs such as parentheses ( ), brackets [ ] and braces { }.

The following punctuation characters . , ; : and ... are used.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| append | bool | byte | cap | close | complex | complex64 | complex128 | uint16 |
| copy | false | float32 | float64 | imag | int | int8 | int16 | uint32 |
| int32 | int64 | iota | len | make | new | nil | panic | uint64 |
| print | println | real | recover | string | true | uint | uint8 | Uintptr |

Go Package, import and visibility

Packages

Packages are used to *categorize* our program so that it can be easy to maintain. Every go-file belongs to some package. Each Go application must have *"main"* package so that it can be compiled.

An application can consist of different packages. Many different .go file can belong to one main package.

We can save Go program by any name but it must have main package. The package name should be written in lowercase letters.

If a package is changed and recompiled, all the client programs that use this package must be recompiled too!

Import

A Go program is linked to different packages through the **import keyword.**

The package names are enclosed within double quotes "". Import loads the public declarations from the compiled package, it does not insert the source code.

We can import multiple packages by a separate statement like:

1. **import** "fmt"
2. **import** "os"

or:

1. **import** "fmt"; **import** "os"

We can also use shorter way like :

1. **import** (
2. "fmt"
3. "os"
4. )

It can be even shorter:

1. **import**("fmt"; "os"

Visibility

An identifier can be variable, constant, function, type or struct field. We can declare identifier in lowercase or uppercase letters.

If we declare identifier in lowercase letter, it will be visible within the package only. But if we declare package in uppercase letter, it will be visible within and outside the package which is also known as exported.

The dot . Operator is used to access the identifier e.g. pack.Age where pack is the package name and Age is the identifier.

**Go controls:**

Go If

The if statement in Go is used to test the condition. If it evaluates to true, the body of the statement is executed. If it evaluates to false, if block is skipped.

Syntax :

1. **if**(boolean\_expression) {
2. /\* statement(s) got executed only if the expression results in true \*/
3. }

**Go if example**

1. **package** main
2. **import** "fmt"
3. func main() {
4. /\* local variable definition \*/
5. var a **int** = 10
6. /\* check the boolean condition using if statement \*/
7. **if**( a % 2==0 ) {      /\* if condition is true then print the following
8. \*/ fmt.Printf("a is even number" )
9. }
10. }

Output:

a is even number

Go if-else

The if-else is used to test the condition. If condition is true, if block is executed otherwise else block is executed.

Syntax :

1. **if**(boolean\_expression) {
2. /\* statement(s) got executed only if the expression results in true \*/
3. } **else** {
4. /\* statement(s) got executed only if the expression results in false \*/
5. }

**Go if-else example**

1. **package** main
2. **import** "fmt"
3. func main() {
4. /\* local variable definition \*/
5. var a **int** = 10;
6. /\* check the boolean condition \*/
7. **if** ( a%2 == 0 ) {
8. /\* if condition is true then print the following \*/
9. fmt.Printf("a is even\n");
10. } **else** {
11. /\* if condition is false then print the following \*/
12. fmt.Printf("a is odd\n");
13. }
14. fmt.Printf("value of a is : %d\n", a);
15. }

Output:

a is even

value of a is : 10

**Go If-else example: with input from user**

1. func main() {
2. fmt.Print("Enter number: ")
3. var input **int**
4. fmt.Scanln(&input)
5. fmt.Print(input)
6. /\* check the boolean condition \*/
7. **if**( input % 2==0 ) {
8. /\* if condition is true then print the following \*/
9. fmt.Printf(" is even\n" );
10. } **else** {
11. /\* if condition is false then print the following \*/
12. fmt.Printf(" is odd\n" );
13. }
14. }

Output:

Enter number: 10

10 is even

Go If else-if chain

The Go if else-if chain is used to execute one statement from multiple conditions.

We can have N numbers of if-else statement. It has no limit.

The curly braces{ } are mandatory in if-else statement even if you have one statement in it. The else-if and else keyword must be on the same line after the closing curly brace }.

**Go If else-if chain Example**



**package** main

**import** "fmt"

func main() {

   fmt.Print("Enter text: ")

   var input **int**

   fmt.Scanln(&input)

**if** (input < 0 || input > 100) {

      fmt.Print("Please enter valid no")

   } **else** **if** (input >= 0 && input < 50  ) {

      fmt.Print(" Fail")

   } **else** **if** (input >= 50 && input < 60) {

      fmt.Print(" D Grade")

   } **else** **if** (input >= 60 && input < 70  ) {

      fmt.Print(" C Grade")

   } **else** **if** (input >= 70 && input < 80) {

      fmt.Print(" B Grade")

   } **else** **if** (input >= 80 && input < 90  ) {

      fmt.Print(" A Grade")

   } **else** **if** (input >= 90 && input <= 100) {

      fmt.Print(" A+ Grade")

   }

}

Output:

Enter text: 84

A Grade

Go Nested if-else

We can also nest the if-else statement to execute one statement from multiple conditions.

Syntax

1. **if**( boolean\_expression 1) {
2. /\* statement(s) got executed only if the expression 1 results in true \*/
3. **if**(boolean\_expression 2) {
4. /\* statement(s) got executed only if the expression 2 results in true \*/
5. }
6. }

**nested if-else example**

1. **package** main
2. **import** "fmt"
3. func main() {
4. /\* local variable definition \*/
5. var x **int** = 10
6. var y **int** = 20
7. /\* check the boolean condition \*/
8. **if**( x >=10 ) {
9. /\* if condition is true then check the following \*/
10. **if**( y >= 10 )  {
11. /\* if condition is true then print the following \*/
12. fmt.Printf("Inside nested If Statement \n" );
13. }
14. }
15. fmt.Printf("Value of x is : %d\n", x );
16. fmt.Printf("Value of y is : %d\n", y );
17. }

Output:

Inside nested If Statement

Value of x is : 10

Value of y is : 20

Go switch

The Go **switch statement** executes one statement from multiple conditions. It is similar to if-else-if chain statement.

Syntax:

1. **switch**  var1 {
2. **case** val1:
3. .....
4. **case** val2
5. .....
6. **default**:
7. .....
8. }

The switch statement in Go is more flexible. In the above syntax, var1 is a variable which can be of any type, and val1, val2, ... are possible values of var1.

In switch statement, more than one values can be tested in a case, the values are presented in a comma separated list

**like:** case val1, val2, val3:

If any case is matched, the corresponding case statement is executed. Here, the break keyword is implicit. So automatic fall-through is not the default behavior in Go switch statement.

For fall-through in Go switch statement, use the keyword "fallthrough" at the end of the branch.

**Go Switch Example:**

**package** main

**import** "fmt"

func main() {

   fmt.Print("Enter Number: ")

  var input **int**

  fmt.Scanln(&input)

**switch** (input) {

**case** 10:

      fmt.Print("the value is 10")

**case** 20:

      fmt.Print("the value is 20")

**case** 30:

      fmt.Print("the value is 30")

**case** 40:

      fmt.Print("the value is 40")

**default**:

      fmt.Print(" It is not 10,20,30,40 ")

   }

}

Output:

Enter Number: 20

the value is 20

**or**

Output:

Enter Number: 35

It is not 10,20,30,40

**Go switch fallthrough example**

1. **import** "fmt"
2. func main() {
3. k := 30
4. **switch** k {
5. **case** 10:
6. fmt.Println("was <= 10"); fallthrough;
7. **case** 20:
8. fmt.Println("was <= 20"); fallthrough;
9. **case** 30:
10. fmt.Println("was <= 30"); fallthrough;
11. **case** 40:
12. fmt.Println("was <= 40"); fallthrough;
13. **case** 50:
14. fmt.Println("was <= 50"); fallthrough;
15. **case** 60:
16. fmt.Println("was <= 60"); fallthrough;
17. **default**:
18. fmt.Println("default case")
19. }
20. }

Output:

was <= 30

was <= 40

was <= 50

was <= 60

default case

Go For Loop

The Go for statement is used for repeating a set of statements number of times. It is the only loop in go language.

There are two variants of for loop in Go: Counter-controlled iteration and Condition-controlled iteration.

When the execution of the loop is over, the objects created inside the loop gets destroyed.

**Go For Loop counter-controlled iteration example:**

**package** main

**import** "fmt"

func main() {

**for** a := 0; a < 11; a++ {

      fmt.Print(a,"\n")

   }

}

Output:

0

1

2

3

4

5

6

7

8

9

10

As you can see in the above example, loop begins with the initialization stage, variable for i (i:= 0); This is done only once. It is followed by a conditional check i (i < 10). Conditional check is performed in every iteration. The for loop stops when the condition becomes false.

**Go Nested For Loop Example:**

**package** main

**import** "fmt"

func main() {

**for** a := 0; a < 3; a++ {

**for** b := 3;b > 0; b-- {

         fmt.Print(a," ",b,"\n")

      }

   }

}

Output:

0 3

0 2

0 1

1 3

1 2

1 1

2 3

2 2

2 1

**Go Infinitive For Loop**

In infinite for loop, the conditional statement is absent like:

1. **for** i:=0; ; i++

**or**

1. **for** { }

Go **Infinitive For Loop Example:**

1. **package** main
2. **import** "fmt"
3. func main() {
4. **for** **true**  {
5. fmt.Printf("This loop will run forever.\n");
6. }
7. }

Output:

This loop will run forever.

This loop will run forever.

This loop will run forever.

This loop will run forever.

This loop will run forever.

This loop will run forever.

This loop will run forever.

**Go For - Condition-controlled iteration**

The for loop which has no header is used for condition-controlled iteration. It is similar to while-loop in other languages.

Syntax:

1. **for** condition { }

**For Loop Example in while fashion:**

**package** main

**import** "fmt"

func main() {

   sum := 1

**for** sum < 100 {

      sum += sum

      fmt.Println(sum)

   }

}

Output:

2

4

8

16

32

64

128

Go for range construct

The for range construct is useful in many context. It can be used to traverse every item in a collection. It is similar to foreach in other languages. But, we still have the index at each iteration in for range construct.

Syntax:

1. **for** ix, val := range coll { }

**Go For Range Example**

**import** "fmt"

func main() {

   nums := []**int**{2, 3, 4}

   sum := 0

**for** \_, value := range nums {// "\_ " is to ignore the index

      sum += value

   }

   fmt.Println("sum:", sum)

**for** i, num := range nums {

**if** num == 3 {

         fmt.Println("index:", I)

      }

   }

   kvs := map[string]string{"1":"mango","2":"apple","3":"banana"}

**for** k, v := range kvs {

      fmt.Printf("%s -> %s\n", k, v)

   }

**for** k := range kvs {

      fmt.Println("key:", k)

   }

**for** i, c := range "Hi" {

      fmt.Println(i, c)

   }

}

Output:

sum: 60

1 -> mango

2 -> apple

3 -> banana

key: 1

key: 2

key: 3

0 72

1 105

Go Goto Statement

The Go goto statement is a jump statement which is used to transfer the control to other parts of the program.

In goto statement, there must be a label. We use label to transfer the control of the program.

**Go Goto Statement Example:**

**package** main

**import** (

   "fmt"

)

func main() {

   var input **int**

Loop:

   fmt.Print("You are not eligible to vote ")

   fmt.Print("Enter your age ")

   fmt.Scanln(&input)

**if** (input <= 17) {

**goto** Loop

   } **else** {

      fmt.Print("You can vote ")

   }

}

Output:

You are not eligible to vote

Enter your age 15

You are not eligible to vote

Enter your age 18

You can vote

Go Break Statement

A break statement is used to break out of the innermost structure in which it occurs. It can be used in for-loop (counter, condition,etc.), and also in a switch. Execution is continued after the ending } of that structure.

Syntax:-

1. **break**;

**Go Break Statement Example:**

**package** main

**import** "fmt"

func main() {

   var  a **int** = 1

**for** a < 10{

      fmt.Print("Value of a is ",a,"\n")

      a++;

**if** a > 5{

         /\* terminate the loop using break statement \*/

**break**;

      }

   }

}

Output:

Value of a is 1

Value of a is 2

Value of a is 3

Value of a is 4

Value of a is 5

Break statement can also be applied in the inner loop, and the control flow break out to the outer loop.

**Go Break Statement with Inner Loop:**

**package** main

**import** "fmt"

func main() {

   var a **int**

   var b **int**

**for** a = 1; a <= 3; a++ {

**for** b = 1; b <= 3; b++ {

**if** (a == 2 && b == 2) {

**break**;

         }

         fmt.Print(a, " ", b, "\n")

     }

   }

}

Output:

1 1

1 2

1 3

2 1

3 1

3 2

3 3

Go Continue Statement

The continue is used to skip the remaining part of the loop, and then continues with the next iteration of the loop after checking the condition.

Syntax:-

1. **continue**;

Or we can do like

1. x:
2. **continue**:x

**Go Continue Statement Example:**

**package** main

**import** "fmt"

func main() {

   /\* local variable definition \*/

   var a **int** = 1

   /\* do loop execution \*/

**for** a < 10 {

**if** a == 5 {

         /\* skip the iteration \*/

         a = a + 1;

**continue**;

      }

      fmt.Printf("value of a: %d\n", a);

      a++;

   }

}

Output:

value of a: 1

value of a: 2

value of a: 3

value of a: 4

value of a: 6

value of a: 7

value of a: 8

value of a: 9

**Continue can be also be applied in the inner loop**

**Go Continue Statement with Inner Loop example:**

1. **package** main
2. **import** "fmt"
3. func main() {
4. /\* local variable definition \*/
5. var a **int** = 1
6. var b **int** = 1
7. /\* do loop execution \*/
8. **for** a = 1; a < 3; a++ {
9. **for** b = 1; b < 3; b++ {
10. **if** a == 2 && b == 2 {
11. /\* skip the iteration \*/
12. **continue**;
13. }
14. fmt.Printf("value of a and b is %d %d\n", a, b);
15. }
16. fmt.Printf("value of a and b is %d %d\n", a, b);
17. }
18. }

Output:

value of a and b is 1 1

value of a and b is 1 2

value of a and b is 1 3

value of a and b is 2 1

value of a and b is 2 3

Go Comments

The Go comments are not executed by the compiler and interpreter. The comments are used to provide information or explanation about the method, variable, class or any statement. It can also be used to hide program code for specific time.

Go Single Line Comment

The double forward slash "//" is used for the single-line comment.

**Go Single Line Comment Example:**

1. **package** main
2. **import** "fmt"
3. func main() {
4. var x **int** = 10 //It is a variable
5. fmt.Print(x)
6. }

Output:

10

Go Multi Line Comment

A multi-line or block-comment starts with /\* and ends with \*/. Here, nesting is not allowed.

**Go Multi Line Comment Example:**

1. **package** main
2. **import** "fmt"
3. func main() {
4. var a **int** = 10
5. /\* Let's declare and
6. print variable in Go\*/
7. fmt.Printf("value of a is %d \n", a);
8. }

Output:

10

# Go Constants

A constant **const** contains data which is not changed. This data can only be of type boolean, number (integer, float or complex) or string.

**Syntax:**

1. **const** identifier [type] = value

**Example:**

1. **const** PI = 3.14159

The type specifier [type] is optional, the compiler can implicitly derive the type from the value.

**Explicit Typing Example:**

1. **const** b string = "abc"

**Implicit Typing Example:**

1. **const** b = "abc"

### **Go Constant Example**

**package** main

**import** "fmt"

func main() {

**const** HEIGHT **int** = 100

**const** WIDTH **int** = 200

   var area **int**

   area = HEIGHT \* WIDTH

   fmt.Printf("value of area : %d", area)

}

Output:

value of area : 20000

o Type Casting

Type casting means conversion of a variable from one data type to another. The value may be lost when large type is converted to a smaller type.

**Go Type Conversion Example**

**package** main

**import** (

   "fmt"

   "strconv"

)

func main() {

   var i **int** = 10

   var f float64 = 6.44

   var str1 string = "101"

   var str2 string = "10.123"

   fmt.Println(float64(i))

   fmt.Println(**int**(f))

   newInt, \_ := strconv.ParseInt(str1, 0, 64)

   fmt.Println(newInt)

   newfloat, \_ := strconv.ParseFloat(str2, 64)

   fmt.Println(newfloat)

}

Output:

10

6

101

10.123

# Go Functions

In Go, functions are the basic building blocks. A function is used to break a large problem into smaller tasks. We can invoke a function several times, hence functions promote code reusability. There are 3 types of functions in Go:

* Normal functions with an identifier
* Anonymous or lambda functions
* Method (A function with a receiver)

Function parameters, return values, together with types, is called function signature.

Function cannot be declared inside another function. If we want to achieve this, we can do this by anonymous function.

### **Go Function Example**

**package** main

**import** "fmt"

type Employee struct {

   fname string

   lname string

}

func (emp Employee) fullname(){

   fmt.Println(emp.fname+" "+emp.lname)

}

func main() {

   e1 := Employee{ "John","Ponting"}

   e1.fullname()

}

Output:

John Ponting

### **Go Function with Return**

Let's see an example of function with return value.

**package** main

**import** (

   "fmt"

)

func fun() **int** {

**return** 123456789

}

func main() {

   x := fun()

   fmt.Println(x)

}

Output:

123

### **Go Function with Multiple Return**

Let's see an example of a function which takes n number of type int as argument and returns two int values. The return values are filled in the calling function in a parallel assignment.

**Go function multiple return example**

**package** main

**import** (

   "fmt"

)

func main() {

   fmt.Println(addAll(10,15,20,25,30))

}

func addAll(args ... **int**)(**int**,**int**)  {

   finalAddValue:=0

   finalSubValue:=0

**for** \_,value  := range args{

      finalAddValue += value

      finalSubValue -= value

   }

**return** finalAddValue,finalSubValue

}

Output:

100 -100

# Go Recursion

In Go programming, calling same function from within the function is known as recursion. It is always a good idea to break a problem into multiple tasks. Let us see a program to calculate factorial value in Go programming using recursion.

### **Go Recursion Example: Factorial Number**

**package** main

**import** (

   "fmt"

)

func main() {

   fmt.Println(factorial(5))

}

func factorial(num **int** ) **int**{

**if** num == 0{

**return** 1

   }

**return** num\*factorial(num-1)

}

Output:

120

# Go Closure

Here, we create an anonymous function which acts as a function closure. A function which has no name is called anonymous function.

A closure is a function which refers reference variable from outside its body. The function may access and assign to the referenced variables.

### **Go Closure Example**

**package** main

**import** (

   "fmt"

)

func main() {

   number := 10

      squareNum := func() (**int**){

      number \*= number

**return** number

   }

   fmt.Println(squareNum())

   fmt.Println(squareNum())

}

Output:

100

10000

# Go Array

In Go, an array is a homogeneous data structure (Fix type) and has a fixed-length. The type can be anything like integers, string or self-defined type.

The items in the array can be accessed through their index, It starts with zero. The number of items in the array is called the length or size of an array. It is fixed and must be declared in the declaration of an array variable.

**Syntax:**

1. var identifier [len]type

**Example:**

1. var arr\_name [5]**int**

### **Go Array Example**

**package** main

**import** "fmt"

func main() {

   var x [5]**int**

   var i, j **int**

**for** i = 0; i < 5; i++ {

      x[i] = i + 10

   }

**for** j = 0; j < 5; j++ {

      fmt.Printf("Element[%d] = %d\n", j, x[j])

   }

}

Output:

Element[0] = 10

Element[1] = 11

Element[2] = 12

Element[3] = 13

Element[4] = 14

## Go Multi Dimensional Arrays

Multi dimensional arrays is simply a list of one-dimensional arrays.

**Sytax:**

1. var arrayName [ x ][ y ] variable\_type

**Example:**

1. a = [3][4]**int**

**Initializing Two-Dimensional Arrays**

1. a = [2][3]**int**{
2. {2, 4, 6} ,   /\*  initializers for row indexed by 0 \*/
3. {8, 10, 12} ,   /\*  initializers for row indexed by 1 \*/
4. }

**Accessing Two-Dimensional Arrays**

1. **int** val = a[1][2]

## Go Multi-Dimensional Array Example

**package** main

**import** "fmt"

func main() {

   /\* an array with 3 rows and 3 columns\*/

   var a = [3][3]**int**{ {1,2,3}, {4,5,6}, {7,8,9}}

   var i, j **int**

   /\* output each array element's value \*/

**for**  i = 0; i < 3; i++ {

**for** j = 0; j < 3; j++ {

         fmt.Print(a[i][j] )

      }

      fmt.Println()

   }

}

Output:

123

456

789

# Go Slice

In Go, slice is a dynamically-sized, segmented view of an underlying array. This segment can be the entire array or a subset of an array. We define the subset of an array by indicating the start and end index. Slices provide a dynamic window onto the underlying array.

**Slice Data Type Example:**

**package** main

**import** (

   "fmt"

)

func main() {

   odd := [6]**int**{2, 4, 6, 8, 10, 12}

   var s []**int** = odd[1:4]

   fmt.Println(s)

}

Output:

[4 6 8]

Slice is like reference to an array. Slice does not store any data. If we change the elements of an array, it will also modify the underlying array. If other slice is referencing the same underlying array, their value will also be changed.

Slice is like array reference. An example of slice is given below:

**package** main

**import** "fmt"

func main() {

   names := [4]string{

      "John",

      "Jim",

      "Jack",

      "jen",

   }

   fmt.Println(names)

   slice1 := names[0:2]

   slice2 := names[1:3]

   fmt.Println(slice1, slice2)

   slice2[0] = "ZZZ"

   fmt.Println(slice1, slice2)

   fmt.Println(names)

}

Output:

[John Jim Jack jen]

[John Jim] [Jim Jack]

[John ZZZ] [ZZZ Jack]

[John ZZZ Jack jen]

### **Slice Literal**

Slice literal is like an array literal without any length. An example of slice without length is given below:

1. **package** main
2. **import** "fmt"
3. func main() {
4. s := []struct {
5. i **int**
6. b bool
7. }{
8. {1, **true**},
9. {2, **false**},
10. {3,**true**},
11. {4, **true**},
12. {5, **false**},
13. {6, **true**},
14. }
15. fmt.Println(s)
16. }

Output:

[{1 true} {2 false} {3 true} {4 true} {5 false} {6 true}]

### **Omit Lower or Upper Bonds**

In slice, we can omit the lower bond or the upper bonds. Zero is the default value of the lower or the upper bond.

1. **package** main
2. **import** "fmt"
3. func main() {
4. slice1 := []**int**{2,4,8,10,12,14}
5. slice2 := slice1[2:4]
6. fmt.Println(slice2)
7. slice3 := slice1[:3]
8. fmt.Println(slice3)
9. slice4 := slice1[2:]
10. fmt.Println(slice4)
11. fmt.Println(slice1)
12. }

Output:

[8 10]

[2 4 8]

[8 10 12 14]

[2 4 8 10 12 14]

### **Slice Length and Capacity**

A slice has length and capacity. The length is the number of stored elements and the capacity is the number of elements of the underlying array counting from the beginning of the slice.

To get the length, we use len(slice) function and to get the capacity, we use cap(slice) function.

**package** main

**import** "fmt"

func main() {

   slice1 := []**int**{2,4,6,8,10,12,14}

   printSlice(slice1)

   // Slice the slice to give it zero length.

   slice2 := slice1[:0]

   printSlice(slice2)

   // Extend its length.

   slice3 := slice1[:4]

   printSlice(slice3)

   // Drop its first two values.

   slice4 := slice1[2:]

   printSlice(slice4)

}

func printSlice(s []**int**) {

   fmt.Printf("length =%d capacity=%d %v\n", len(s), cap(s), s)

}

Output:

length =7 capacity=7 [2 4 6 8 10 12 14]

length =0 capacity=7 []

length =4 capacity=7 [2 4 6 8]

length =5 capacity=5 [6 8 10 12 14]

### **Slice Make Function**

We can also create slice with the help of make function. The make function creates a zero sized array and returns slice of the array.

1. **package** main
2. **import** "fmt"
3. func main() {
4. slice := make([]**int**, 10)
5. printSlice("slice", slice)
6. slice1 := make([]**int**, 0, 10)
7. printSlice("slice1", slice1)
8. slice2 := slice1[:5]
9. printSlice("slice2", slice2)
10. slice3 := slice2[2:5]
11. printSlice("slice3", slice3)
12. }
13. func printSlice(s string, x []**int**) {
14. fmt.Printf("%s length=%d capacity=%d %v\n",
15. s, len(x), cap(x), x)
16. }

Output:

slice length=10 capacity=10 [0 0 0 0 0 0 0 0 0 0]

slice1 length=0 capacity=10 []

slice2 length=5 capacity=10 [0 0 0 0 0]

slice3 length=3 capacity=8 [0 0 0]

Go Command Line Arguments

When we need to execute a program with some arguments, we generally use command line argument. The arguments passed from the console can be received by the Go program and it can be used as an input.

The os.Args is used to get the arguments. The index 0 of os.Args contains the path of the program.

The os.Args[1:] holds provided arguments.

Go Command Line Argument Example 1

1. **package** main
3. **import** (
4. "fmt"
5. "os"
6. )
7. func main() {
8. var s, arg string
9. **for** i := 1; i < len(os.Args); i++ {
10. s += arg + os.Args[i]+" "
11. }
12. fmt.Println(s)
13. }

**Command to run the program:**

1. go build ProgramName.go
2. ./ProgramName Tom Dick Harry

**Output:**

1. Tom Dick Harry

Go Command Line Argument Example 2

1. **package** main
2. **import** "os"
3. **import** "fmt"
4. func main() {
5. arumentWithPath := os.Args //returns all arguments including path
6. arumentSlice:= os.Args[1:] //returns all elements after path
7. arumentAt2 := os.Args[2] //returns specified argument only
8. fmt.Println(arumentWithPath)
9. fmt.Println(arumentSlice)
10. fmt.Println(arumentAt2)
11. }

Output:

[/private/var/folders/by/w452m9913bj8\_rmgzgjsvw9w0000gn/T/\_\_\_cmd\_go Tom Dick Harry]

[Tom Dick Harry]

Dick

# Go Strings

The Go string is a sequence of variable-width characters.

Go strings and text files occupy less memory or disk space. Since, UTF-8 is the standard, Go doesn't need to encode and decode strings.

Go Strings are value types and immutable. It means if you create a string, you cannot modify the contents of the string. The initial value of a string is empty "" by default.

### **Go String Example**

**package** main

**import** ("fmt"

      "reflect"

)

func main()  {

   var x string = "Hello World"

   fmt.Println(x)

   fmt.Println(reflect.TypeOf(x))

}

Output:

Hello World

string

### **Go String len() Example**

**package** main

**import** "fmt"

func main() {

   str := "I love my country"

   fmt.Println(len(str))

}

Output:

17

### **Go Print ASCII Example**

**package** main

**import** "fmt"

func main() {

   fmt.Println("Ascii value of A is ","A"[0])

}

Output:

Ascii value of A is 65

### **Go String ToUpper() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   str := "india"

   fmt.Println(strings.ToUpper(str))

}

Output:

INDIA

### **Go String ToLower() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   str := "INDIA"

   fmt.Println(strings.ToLower(str))

}

Output:

india

### **Go String HasPrefix() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   s := "INDIA"

   fmt.Println(strings.HasPrefix(s,"IN"))

}

Output:

true

### **Go String HasSuffix() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   s := "INDIA"

   fmt.Println(strings.HasSuffix(s,"IA"))

}

Output:

true

### **Go String Join() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   var arr = []string{"a","b","c","d"}

   fmt.Println(strings.Join(arr,"\*"))

}

Output:

a\*b\*c\*d

### **Go String Repeat() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   var str = "New "

   fmt.Println(strings.Repeat(str,4))

}

Output:

New New New New

### **Go String Contains() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   str:= "Hi...there"

   fmt.Println(strings.Contains(str,"th"))

}

Output:

true

### **Go String Index() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   str:= "Hi...there"

   fmt.Println(strings.Index(str,"th"))

}

Output:

5

### **Go String Count() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   str:= "Hi...there"

   fmt.Println(strings.Count(str,"e"))

}

Output:

2

### **Go String Replace() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   str:= "Hi...there"

   fmt.Println(strings.Replace(str,"e","Z",2))

}

Output:

Hi...thZrZ

### **Go String Split() Example**

**package** main

**import** "fmt"

**import** "strings"

func main() {

   str := "I,love,my,country"

   var arr []string = strings.Split(str, ",")

   fmt.Println(len(arr))

**for** i, v := range arr {

      fmt.Println("Index : ", i, "value : ", v)

   }

}

Output:

4

Index : 0 value : I

Index : 1 value : love

Index : 2 value : my

Index : 3 value : country

### **Go String Split() Example 2**

**package** main

**import** (

   "fmt"

   "strings"

)

func main() {

   fmt.Printf("%q\n", strings.Split("x,y,z", ","))

   fmt.Printf("%q\n", strings.Split(" John and Jack and Johnny and Jinn ", "and"))

   fmt.Printf("%q\n", strings.Split(" abc ", ""))

   fmt.Printf("%q\n", strings.Split("", "Hello"))

}

Output:

["x" "y" "z"]

[" John " " Jack " " Johnny " " Jinn"]

[" " "a" "b" "c" " "]

[""]

### **Go String Compare() Example**

**package** main

**import** (

   "fmt"

   "strings"

)

func main() {

   fmt.Println(strings.Compare("a", "b"))

   fmt.Println(strings.Compare("a", "a"))

   fmt.Println(strings.Compare("b", "a"))

}

Output:

-1

0

1

### **Go String Trim() Example**

**package** main

**import** (

   "fmt"

   "strings"

)

func main() {

      fmt.Println(strings.TrimSpace(" \t\n I love my country  \n\t\r\n"))

}

Output:

I love my country

### **Go String ContainsAny() Example**

**package** main

**import** (

   "fmt"

   "strings"

)

func main() {

   fmt.Println(strings.ContainsAny("Hello", "A"))

   fmt.Println(strings.ContainsAny("Hello", "o & e"))

   fmt.Println(strings.ContainsAny("Hello", ""))

   fmt.Println(strings.ContainsAny("", ""))

}

Output:

false

true

false

false

# Go Regex

Go Regex package is used for searching string. To search a string, we need to provide a pattern of the string.

We need to compile the pattern into the regex object so that we can invoke methods through it.

The regex object can be retrieved by using compile() and mustcompile() functions. Now we can use functions to find strings such as FindString(), FindStringSubmatch(), FindStringIndex() etc.

### **Go Regex Example 1**

**package** main

**import** (

    "fmt"

    "regexp"

)

func main() {

    re := regexp.MustCompile(".com")

    fmt.Println(re.FindString("google.com"))

    fmt.Println(re.FindString("abc.org"))

    fmt.Println(re.FindString("fb.com"))

}

Output:

.com

.com

The FindString() method returns a string having the text of the left most match. If no match is found, empty string is returned.

### **Go Regex Example 2**

**package** main

**import** (

    "fmt"

    "regexp"

)

func main() {

    re := regexp.MustCompile(".com")

    fmt.Println(re.FindStringIndex("google.com"))

    fmt.Println(re.FindStringIndex("abc.org"))

    fmt.Println(re.FindStringIndex("fb.com"))

}

Output:

[6 10]

[]

[2 6]

### **Go Regex Example 3**

we can also use FindStringSubmatch() method which returns a slice of strings having the text of the leftmost match and the matches. If no match is found, the return value is an empty string.

**package** main

**import** (

    "fmt"

    "regexp"

)

func main() {

    re := regexp.MustCompile("f([a-z]+)ing")

    fmt.Println(re.FindStringSubmatch("flying"))

    fmt.Println(re.FindStringSubmatch("abcfloatingxyz"))

}

Output:

[flying ly]

[floating loat]

Process finished with exit code 0

Go Struct

In Go, Struct can be used to create user-defined types.

Struct is a composite type means it can have different properties and each property can have their own type and value.

Struct can represent real-world entity with these properties. We can access a property data as a single entity. It is also valued types and can be constructed with the new() function.

Go Struct Example

**package** main

**import** (

   "fmt"

)

type person struct {

   firstName string

   lastName  string

   age       **int**

}

func main() {

   x := person{age: 30, firstName: "John", lastName: "Anderson", }

   fmt.Println(x)

   fmt.Println(x.firstName)

}

Output:

{John Anderson 30}

John

Types: Go Embedded Struct

Struct is a data type and can be used as an anonymous field (having only the type). One struct can be inserted or "embedded" into other struct.

It is a simple 'inheritance' which can be used to implement implementations from other type or types.

Go Embedded Struct Example

**package** main

**import** (

   "fmt"

)

type person struct {

   fname string

   lname string}

type employee struct {

   person

   empId **int**

}

func (p person) details() {

   fmt.Println(p, " "+" I am a person")

}

func (e employee) details() {

   fmt.Println(e, " "+"I am a employee")

}

func main() {

   p1 := person{"Raj", "Kumar"}

   p1.details()

   e1 := employee{person:person{"John", "Ponting"}, empId: 11}

   e1.details()

}

Output:

{Raj Kumar} I am a person

{{John Ponting} 11} I am a employee

# Go Interface

Go has different approaches to implement the concepts of object-orientation. Go does not have classes and inheritance. Go fulfill these requirements through its powerful interface.

Interfaces provide behavior to an object: if something can do this, then it can be used here.

An interface defines a set of abstract methods and does not contain any variable.

syntax:

1. type Namer **interface** {
2. Method1(param\_list) return\_type
3. Method2(param\_list) return\_type
4. ...
5. }

where Namer is an interface type.

Generally, the name of an interface is formed by the method name plus the [e]r suffix, such as Printer, Reader, Writer, Logger, Converter, etc.

* A type doesn't have to state explicitly that it implements an interface: interfaces are satisfied implicitly. Multiple types can implement the same interface.
* A type that implements an interface can also have other functions.
* A type can implement many interfaces.
* An interface type can contain a reference to an instance of any of the types that implement the interface

### **Go Interface Example**

**package** main

**import** (

   "fmt"

)

type vehicle **interface** {

   accelerate()

}

func foo(v vehicle)  {

   fmt.Println(v)

}

type car struct {

   model string

   color string

}

func (c car) accelerate()  {

   fmt.Println("Accelrating?")

}

type toyota struct {

  model string

   color string

   speed **int**

}

func (t toyota) accelerate(){

   fmt.Println("I am toyota, I accelerate fast?")

}

func main() {

   c1 := car{"suzuki","blue"}

   t1:= toyota{"Toyota","Red", 100}

   c1.accelerate()

   t1.accelerate()

   foo(c1)

   foo(t1)

}

Output:

Accelrating...

I am toyota, I accelerate fast...

{suzuki blue}

{Toyota Red 100}

# Go Pointer

A pointer is a variable that stores the address of another variable. The general form of a pointer variable declaration is:

1. var var\_name \*var-type

A newly declared pointer which has not been assigned to a variable has the nil value.

The address-of operator &, when placed before a variable gives us the memory address of the variable.

With pointers, we can pass a reference to a variable (for example, as a parameter to a function), instead of passing a copy of the variable which can reduce memory usage and increase efficiency.

### **Go Pointer Example 1**

**package** main

**import** (

   "fmt"

)

func main() {

   x:=10

   changeX(&x)

   fmt.Println(x)

}

func changeX(x \***int**){

   \*x=0

}

Output:

x = 0

### **Go Pointer Example 2**

**package** main

**import** (

   "fmt"

)

func main() {

   ptr := **new**(**int**)

   fmt.Println("Before change ptr",\*ptr)

   changePtr(ptr)

   fmt.Println("After change ptr",\*ptr)

}

func changePtr(ptr \***int**)  {

   \*ptr = 10

}

Output:

Before change ptr 0

After change ptr 10

Go Reflect

Go Reflection is the ability of a program to examine its own structure, particularly through the types; it's a form of meta-programming.

Reflect can be used to investigate types and variables at runtime, e.g. its size, its methods, and it can also call these methods 'dynamically'.

**Go Reflect example**

**package** main

**import**(

   "fmt"

   "reflect"

)

func main()  {

   age := 27.5

   fmt.Printf("%T\n" ,age)

   fmt.Println(reflect.TypeOf(age))

}

Output:

float64

float64

Go Rune Type

The type rune is an alias for type int32. The rune literals are an integer value.

If you store string value in rune literal, it will provide the ASCII value of the character. For example, the rune literal of 'A' will be 65.

**Go Rune Type Example**

**package** main

**import**(

   "fmt"

   "reflect"

)

func main(){

   rune := 'A'

   fmt.Printf("%d \n", rune)

   fmt.Println(reflect.TypeOf(rune))

}

Output:

65

int32

Go Map

In Go, Maps is an unordered collection of key and its associated value. They are very good for looking up values fast. The key type must have the operations == and != defined, like string, int, float.

Hence arrays, slices and structs cannot be used as key type, but pointers and interface types can.

structs can be used as a key when we provide Key() or Hash() method, so that a unique numeric or string key can be calculated from the struct's fields.

A map is a reference type and declared in general as:

1. var map1 map[keytype]valuetype

**e.g.**

1. var map1 map[**int**]string

Go Map Example

**package** main

**import** "fmt"

func main ()  {

   x := map[string]**int**{"Kate":28,"John":37, "Raj":20}

   fmt.Print(x)

   fmt.Println("\n",x["Raj"])

}

Output:

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

20

Go Map Insert and Update operation

Update and insert operation are similar in go map. If the map does not contain the provided key the insert operation will takes place and if the key is present in the map then update operation takes place.

**package** main

**import** "fmt"

func main() {

   m := make(map[string]**int**)

   fmt.Println(m)

   m["Key1"] = 10

   m["Key2"] = 20

   m["Key3"] = 30

   fmt.Println(m)

   m["Key2"] = 555

   fmt.Println(m)

}

Output:

map[]

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

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

Go Map Delete operation

You can delete an element in Go Map using delete() function.

Syntax:

1. delete(map, key)

Example:

**package** main

**import** "fmt"

func main() {

   m := make(map[string]**int**)

   m["Key1"] = 10

   m["Key2"] = 20

   m["Key3"] = 30

   fmt.Println(m)

   delete(m, "Key3")

   fmt.Println(m)

}

Output:

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

map[Key2:20 Key1:10]

Go Map Retrieve Element

Syntax:

1. elem = m[key]

Example:

**package** main

**import** "fmt"

func main() {

   m := make(map[string]**int**)

   m["Key1"] = 10

   m["Key2"] = 20

   m["Key3"] = 30

   fmt.Println(m)

   fmt.Println("The value:", m["Key2"])

}

Output:

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

The value: 20

We can also test if a key is present in the table with two value example

syntax:

1. elem, ok = m[key]

If key is not present, then the value of elem is the default value of element type.

If the type of elem is int then value of elem is zero.

**package** main

**import** "fmt"

func main() {

   m := make(map[string]**int**)

   m["Key1"] = 10

   m["Key2"] = 20

   m["Key3"] = 30

   fmt.Println(m)

   v, ok := m["Key2"]

   fmt.Println("The value:", v, "Present?", ok)

   i, j := m["Key4"]

   fmt.Println("The value:", i, "Present?", j)

}

Output:

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

The value: 20 Present? true

The value: 0 Present? false

In Go, Maps are like struct, but it requires keys

Go Map of Struct

**package** main

**import** "fmt"

type Vertex struct {

   Latitude, Longitude float64

}

var m = map[string]Vertex{

   "JavaTpoint": Vertex{     40.68433, -74.39967,   },

   "SSS-IT": Vertex{     37.42202, -122.08408,  },

}

func main() {

   fmt.Println(m)

}

Output:

map[JavaTpoint:{40.68433 -74.39967} SSS-IT:{37.42202 -122.08408}]

# Go Error

Go does not have an exception mechanism like try/catch in Java, we cannot throw an exception in Go.

Go uses a different mechanism which is known as defer-panic-and-recover mechanism.

Go handles simple errors for function, methods by returning an error object. The error object may be the only or the last return value. The error object is nil if there is no error in the function.

We should always check the error at the calling statement, if we receive any of it or not.

We should never ignore errors, it may lead to program crashes.

The way go detect and report the error condition is

* A function which can result in an error returns two variables: a value and an error-code which is nil in case of success, and != nil in case of an error-condition.
* The error is checked, after the function call . In case of an error ( if error != nil), the execution of the actual function (or if necessary the entire program) is stopped.

Go has predefined error interface type

1. type error **interface** {
2. Error() string
3. }

We can define error type by using error.New from the error package and provide an appropriate error message like:

1. err := errors.New("math - square root of negative number")

### **Go Error Example**

**package** main

**import** "errors"

**import** "fmt"

**import** "math"

func Sqrt(value float64) (float64, error) {

**if** (value < 0) {

**return** 0, errors.New("Math: negative number passed to Sqrt")

   }

**return** math.Sqrt(value), nil

}

func main() {

   result, err := Sqrt(-64)

**if** err != nil {

      fmt.Println(err)

   } **else** {

      fmt.Println(result)

   }

   result, err = Sqrt(64)

**if** err != nil {

      fmt.Println(err)

   } **else** {

      fmt.Println(result)

   }

}

Output:

Math: negative number passed to Sqrt

8

# Go Recover

Recover is used to regain control of a program from panic or error-condition. It stops the terminating sequence and resumes the normal execution. It is called from the deferred function. It retrieves the error value passed through the call of panic. Normally, it returns **nil** which has no other effect.

### **Go recover() Example**

**package** main

**import** (

   "fmt"

)

func main() {

   fmt.Println(SaveDivide(10, 0))

   fmt.Println(SaveDivide(10, 10))

}

func SaveDivide(num1, num2 **int**) **int** {

   defer func() {

      fmt.Println(recover())

   }()

   quotient := num1 / num2

**return** quotient

}

Output:

runtime error: integer divide by zero

0

# Go defer keyword

The defer keyword is generally used for cleaning purpose. The defer keyword postpones the execution of a function or statement until the end of the calling function.

It executes code (a function or expression) when the enclosing function returns before the closing curly brace }. It is also executed if an error occurs during the execution of the enclosing function.

### **Go defer Example**

**package** main

**import** (

   "fmt"

)

func main() {

   defer print1("Hi...")

   print2("there")

}

func  print1(s string)  {

   fmt.Println(s)

}

func print2(s string)  {

   fmt.Println(s)

}

Output:

there

Hi...

# Go Panic

Go panic is a mechanism by which we handle error situations. Panic can be used to abort a function execution. When a function calls panic, its execution stops and the control flows to the associated deferred function.

The caller of this function also gets terminated and caller's deferred function gets executed (if present any). This process continues till the program terminates. Now the error condition is reported.

This termination sequences is called panicking and can be controlled by the built-in function recover.

### **Go Panic Example 1:**

**package** main

**import** "os"

func main() {

    panic("Error Situation")

    \_, err := os.Open("/tmp/file")

**if** err != nil {

        panic(err)

    }

}

Output:

panic: Error Situation

goroutine 1 [running]:

main.main()

/Users/pro/GoglandProjects/Panic/panic example1.go:6 +0x39

### **Go Panic Example 2**

**package** main

**import** "fmt"

func main() {

    fmt.Println("Calling x from main.")

    x()

    fmt.Println("Returned from x.")

}

func x() {

    defer func() {

**if** r := recover(); r != nil {

            fmt.Println("Recovered in x", r)

        }

    }()

    fmt.Println("Executing x...")

    fmt.Println("Calling y.")

    y(0)

    fmt.Println("Returned normally from y.")

}

func y(i **int**) {

    fmt.Println("Executing y....")

**if** i > 2 {

        fmt.Println("Panicking!")

        panic(fmt.Sprintf("%v" , i))

    }

    defer fmt.Println("Defer in y", i)

    fmt.Println("Printing in y", i)

    y(i + 1)

}

Output:

Calling x from main.

Executing x...

Calling y.

Executing y....

Printing in y 0

Executing y....

Printing in y 1

Executing y....

Printing in y 2

Executing y....

Panicking!

Defer in y 2

Defer in y 1

Defer in y 0

Recovered in x 3

Returned from x.

Go Concurrency

Large programs are divided into smaller sub-programs. Programs which run their smaller components at the same time is known as concurrency.

Goroutines

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; This can be done by sync package, but it is recommended to use channels to synchronize goroutines.

A goroutine is implemented as a function or method. It is called (invoked) with the 'go' keyword. When the goroutine finishes, nothing is returned to the caller function.

Go Concurrency Example

**package** main

**import** (

   "fmt"

   "time"

   "sync"

)

var wg = sync.WaitGroup{}

func main() {

   wg.Add(2)

   go fun1()

   go fun2()

   wg.Wait()

}

func fun1(){

**for**  i:=0;i<10;i++{

      fmt.Println("fun1,  ->",i)

      time.Sleep(time.Duration(5\*time.Millisecond))

   }

   wg.Done()

}

func fun2(){

**for** i:=0;i<10;i++{

      fmt.Println("fun2,  ->",i)

      time.Sleep(time.Duration(10\*time.Millisecond))

   }

   wg.Done()

}

Output:

fun2, -> 0

fun1, -> 0

fun1, -> 1

fun2, -> 1

fun1, -> 2

fun1, -> 3

fun2, -> 2

fun1, -> 4

fun1, -> 5

fun2, -> 3

fun1, -> 6

fun1, -> 7

fun2, -> 4

fun1, -> 8

fun1, -> 9

fun2, -> 5

fun2, -> 6

fun2, -> 7

fun2, -> 8

fun2, -> 9

Go Race

A race condition occurs in Go when two or more goroutines try to access the same resource. It may happen when a variable attempts to read and write the resource without any regard to other routines.

Go Race Condition Example

**package** main

**import** (

   "sync"

   "time"

   "math/rand"

   "fmt"

)

var wait sync.WaitGroup

var count **int**

func  increment(s string)  {

**for** i :=0;i<10;i++ {

      x := count

      x++;

      time.Sleep(time.Duration(rand.Intn(4))\*time.Millisecond)

      count = x;

      fmt.Println(s, i,"Count: ",count)

   }

   wait.Done()

}

func main(){

   wait.Add(2)

   go increment("foo: ")

   go increment("bar: ")

   wait.Wait()

   fmt.Println("last count value " ,count)

}

Output:

foo: 0 Count: 1

bar: 0 Count: 1

foo: 1 Count: 2

foo: 2 Count: 3

foo: 3 Count: 4

bar: 1 Count: 2

foo: 4 Count: 5

foo: 5 Count: 6

foo: 6 Count: 7

bar: 2 Count: 3

bar: 3 Count: 4

bar: 4 Count: 5

foo: 7 Count: 8

foo: 8 Count: 9

bar: 5 Count: 6

bar: 6 Count: 7

foo: 9 Count: 10

bar: 7 Count: 8

bar: 8 Count: 9

bar: 9 Count: 10

last count value 10

As you can see in the above example, the count resource is accessed by 2 go routines. Each routine iterates to 10 times. In such case, the count variable should be 20 at last. But it is not so because it is simulating race condition.

Go Mutex

Mutual Exclusion locks, or mutexes can be used to synchronize access to state and safely access data across many goroutines. It acts as a guard to the entrance of the critical section of code so that only one thread can enter the critical section at a time.

We set a lock around particular lines of code with it. While one Goroutine holds the lock, all other Goroutines are prevented from executing any lines of code protected by the same mutex, and are forced to wait until the lock is yielded before they can proceed.

**Go Mutex Example**

**package** main

**import** (

   "sync"

   "time"

   "math/rand"

   "fmt"

)

var wait sync.WaitGroup

var count **int**

var mutex sync.Mutex

func  increment(s string)  {

**for** i :=0;i<10;i++ {

      mutex.Lock()

      x := count

      x++;

      time.Sleep(time.Duration(rand.Intn(10))\*time.Millisecond)

      count = x;

      fmt.Println(s, i,"Count: ",count)

      mutex.Unlock()

   }

   wait.Done()

}

func main(){

   wait.Add(2)

   go increment("foo: ")

   go increment("bar: ")

   wait.Wait()

   fmt.Println("last count value " ,count)

}

Output:

bar: 0 Count: 1

bar: 1 Count: 2

bar: 2 Count: 3

bar: 3 Count: 4

bar: 4 Count: 5

bar: 5 Count: 6

bar: 6 Count: 7

bar: 7 Count: 8

bar: 8 Count: 9

bar: 9 Count: 10

foo: 0 Count: 11

foo: 1 Count: 12

foo: 2 Count: 13

foo: 3 Count: 14

foo: 4 Count: 15

foo: 5 Count: 16

foo: 6 Count: 17

foo: 7 Count: 18

foo: 8 Count: 19

foo: 9 Count: 20

last count value 20

# Go Atomic Variable

Atomic variables are used to manage state, though sync/atomic package and avoid race conditions. Atomic counters can be accessed by multiple go routines.

### **Go Atomic Variable Example**

**package** main

**import** (

   "sync"

   "time"

   "math/rand"

   "fmt"

   "sync/atomic"

)

var wait sync.WaitGroup

var count int64

func  increment(s string)  {

**for** i :=0;i<10;i++ {

      time.Sleep(time.Duration((rand.Intn(3)))\*time.Millisecond)

      atomic.AddInt64(&count,1)

      fmt.Println(s,i,"Count ->",count)

   }

   wait.Done()

}

func main(){

   wait.Add(2)

   go increment("foo: ")

   go increment("bar: ")

   wait.Wait()

   fmt.Println("last count value " ,count)

}

Output:

foo: 0 Count -> 1

foo: 1 Count -> 2

bar: 0 Count -> 3

bar: 1 Count -> 4

bar: 2 Count -> 5

foo: 2 Count -> 6

bar: 3 Count -> 7

bar: 4 Count -> 8

bar: 5 Count -> 9

foo: 3 Count -> 10

bar: 6 Count -> 11

bar: 7 Count -> 12

foo: 4 Count -> 13

foo: 5 Count -> 14

bar: 8 Count -> 15

bar: 9 Count -> 16

foo: 6 Count -> 17

foo: 7 Count -> 18

foo: 8 Count -> 19

foo: 9 Count -> 20

last count value 20

# Go Channel

The channel acts as a pipe by which we send typed values from one Goroutine to another. It guarantees synchronization since only one Goroutine has access to a data item at any given time. The ownership of the data is passed between different Goroutine. Hence, By design it avoids the pitfalls of shared memory and prevent race condition.

### **Go Channel Example**

**package** main

**import** "fmt"

**import** "time"

func worker(done chan bool) {

   fmt.Print("working...")

   time.Sleep(time.Second)

   fmt.Println("done")

   done <- **true**

}

func main() {

   done := make(chan bool, 1)

   go worker(done)

   <-done

}

Output:

working...done

Go Worker Pools

Worker pools is a design in which a fixed number of m workers (Go goroutines) works on n tasks in a work queue (Go channel). The work resides in a queue until a worker finish its current task and pull a new one.

Let's See it by an example

**package** main

**import** (

"fmt"

"time"

)

func worker(id **int**, jobs <-chan **int**, results chan<- **int**) {

**for** j := range jobs {

        fmt.Println("worker", id, "processing job", j)

        time.Sleep(time.Second)

        results <- j \* 2

    }

}

func main() {

    job := make(chan **int**, 10)

    result := make(chan **int**, 10)

**for** w := 1; w <= 2; w++ {

        go worker(w, job, result)

    }

**for** j := 1; j <= 9; j++ {

        job <- j

    }

    close(job)

**for** a := 1; a <= 9; a++ {

        <-result

    }

}

Output:

worker 2 processing job 1

worker 1 processing job 2

worker 2 processing job 3

worker 1 processing job 4

worker 1 processing job 5

worker 2 processing job 6

worker 1 processing job 7

worker 2 processing job 8

In this example, 2 workers are started and 9 work items are in put onto a job channel. Workers have a work loop with a time.Sleep so that each ends up working 2 jobs. close is used on the channel after all the work's been put onto it, which signals to all 2 workers that they can exit their work loop by dropping them out of their loop on range.

Go File I/O

In go os.file objects are used for file manipulations. os.File objects are also called filehandles.

open function which is in os package is used to open files in Go. ReadFile() in the io/ioutil package is used to read the file. This method returns []byte array of read bytes. file.WriteString method can be used to write to the file.

We use defer file.close() right after opening the file to make sure that the file is closed as soon as the function completes. If a file does not exists or the program has not the sufficient rights to open the file then

inputFile, inputError = os.Open("input.dat") results in an error.

**Go File I/O Example**

**package** main

**import** (

   "os"

   "log"

   "io/ioutil"

   "fmt"

)

func main() {

   file, err := os.Create("file.txt")

**if** err != nil {

      log.Fatal(err)

   }

   file.WriteString("Hi... there")

   file.Close()

   stream, err:= ioutil.ReadFile("file.txt")

**if** err != nil {

      log.Fatal(err)

   }

   readString := string(stream)

   fmt.Println(readString)

}

Output:

Hi... there

Go HTTP Server

Go can be also used to create web applications. Net/http is a library package used to build web applications. It has HandelFunc() function which routes the incoming request to its corresponding function. The ListenAndServe function is used to create a resource server which listens to the provided port. The function someFunc has the http.ResponceWriter and http.Request type parameter. It is responsible to take the incoming request and after processing the return response.

**Go http server example**

**package** main

**import** (

   "fmt"

   "net/http"

)

func main() {

   http.HandleFunc("/",MyHandler1)

   http.HandleFunc("/John",MyHandler2)

   http.ListenAndServe(":8080",nil)

}

func MyHandler1(w http.ResponseWriter,r \*http.Request){

   fmt.Fprint(w,"Hello World\n")

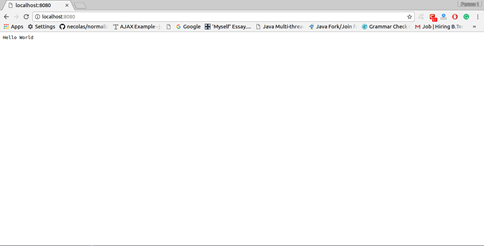
}

func MyHandler2(w http.ResponseWriter,r \*http.Request){

   fmt.Fprint(w,"Hello John\n")

}

Output:



# Go URL Parsing

Go has good support for url parsing. URL contains a scheme, authentication info, host, port, path, query params, and query fragment. we can parse URL and deduce what are the parameter is coming to the server and then process the request accordingly.

The net/url package has the required functions like Scheme, User, Host, Path, RawQuery etc.

### **Go URL Parsing Example 1**

**package** main

**import** "fmt"

**import** "net"

**import** "net/url"

func main() {

    p := fmt.Println

    s := "Mysql://admin:password@serverhost.com:8081/server/page1?key=value&key2=value2#X"

    u, err := url.Parse(s)

**if** err != nil {

        panic(err)

    }

    p(u.Scheme) //prints Schema of the URL

    p(u.User)   // prints the parsed user and password

    p(u.User.Username())    //prints user's name

    pass, \_ := u.User.Password()

    p(pass)     //prints user password

    p(u.Host)       //prints host and port

    host, port, \_ := net.SplitHostPort(u.Host)       //splits host name and port

    p(host)     //prints host

    p(port)     //prints port

    p(u.Path)       //prints the path

    p(u.Fragment)       //prints fragment path value

    p(u.RawQuery)       //prints query param name and value as provided

    m, \_ := url.ParseQuery(u.RawQuery)      //parse query param into map

    p(m)        //prints param map

    p(m["key2"][0])     //prints specific key value

}

Output:

mysql

admin:password

admin

password

serverhost.com:8081

serverhost.com

8081

/server/page1

X

key=value&key2=value2

map[key:[value] key2:[value2]]

value2

### **Go URL Parsing Example 2**

**package** main

**import** (

    "io"

    "net/http"

)

func main() {

    http.HandleFunc("/company", func(res http.ResponseWriter, req \*http.Request) {

        displayParameter(res, req)

    })

    println("Enter the url in browser:  http://localhost:8080/company?name=Tom&age=27")

    http.ListenAndServe(":8080", nil)

}

func displayParameter(res http.ResponseWriter, req \*http.Request) {

    io.WriteString(res, "name: "+req.FormValue("name"))

    io.WriteString(res, "\nage: "+req.FormValue("age"))

}

Output:

Enter the url in browser: <http://localhost:8080/company?name=Tom&age=27>

# Go Url Passing 1 Go REST API Example



**package** main

**import** (

   "encoding/json"

   "log"

   "net/http"

   "github.com/gorilla/mux"

)

type Employee struct {

   ID        string   'json:"id,omitempty"'

   Firstname string   'json:"firstname,omitempty"'

   Lastname  string   'json:"lastname,omitempty"'

   Address   \*Address 'json:"address,omitempty"'

}

type Address struct {

   City  string 'json:"city,omitempty"'

   State string 'json:"state,omitempty"'

}

var emp []Employee

func GetEmpIdEndpoint(w http.ResponseWriter, req \*http.Request) {

   params := mux.Vars(req)

**for** \_, item := range emp {

**if** item.ID == params["id"] {

         json.NewEncoder(w).Encode(item)

**return**

      }

   }

   json.NewEncoder(w).Encode(&Employee{})

}

func GetEmployeeEndpoint(w http.ResponseWriter, req \*http.Request) {

   json.NewEncoder(w).Encode(emp)

}

func CreateEmployeeEndpoint(w http.ResponseWriter, req \*http.Request) {

   params := mux.Vars(req)

   var person Employee

   \_ = json.NewDecoder(req.Body).Decode(&person)

   person.ID = params["id"]

   emp = append(emp, person)

   json.NewEncoder(w).Encode(emp)

}

func DeleteEmployeeEndpoint(w http.ResponseWriter, req \*http.Request) {

   params := mux.Vars(req)

**for** index, item := range emp {

**if** item.ID == params["id"] {

         emp = append(emp[:index], emp[index+1:]...)

**break**

      }

   }

   json.NewEncoder(w).Encode(emp)

}

func main() {

   router := mux.NewRouter()

   emp = append(emp, Employee{ID: "1", Firstname: "Nic", Lastname: "Raboy",

   Address: &Address{City: "Dublin", State: "CA"}})

   emp = append(emp, Employee{ID: "2", Firstname: "Maria", Lastname: "Raboy"})

   router.HandleFunc("/employee", GetEmployeeEndpoint).Methods("GET")

   router.HandleFunc("/employee/{id}", GetEmpIdEndpoint).Methods("GET")

   router.HandleFunc("/employee/{id}", CreateEmployeeEndpoint).Methods("POST")

   router.HandleFunc("/employee/{id}", DeleteEmployeeEndpoint).Methods("DELETE")

   log.Fatal(http.ListenAndServe(":12345", router))

}

Output:

You can check the output by installing postman extension for chrome browser

**Method --> Get,**

1. url -->http://localhost:8080/employee
2. Response :
3. [{"id":"1","firstname":"James","lastname":"Johnson","address":{"city":"Hoseynabad","state":"Kavir"}}
4. ,{"id":"2","firstname":"Sarah","lastname":"Taylor","address":{"city":"Kamenka","state":"Vyborgsky"}}]

**Method --> GET,**

1. url -->http://localhost:8080/employee/1
2. Response :
3. {"id":"1","firstname":"James","lastname":"Johnson","address":
4. "city":"Hoseynabad","state":"Kavir"}}

**Method --> POST,**

1. url -->http://localhost:8080/employee/3
2. Response :
3. [{"id":"1","firstname":"James","lastname":"Johnson","address":{"city":"Hoseynabad","state":"Kavir"}},
4. {"id":"2","firstname":"Sarah","lastname":"Taylor","address":{"city":"Kamenka","state":"Vyborgsky"}},
5. {"id":"3","firstname":"Roger","lastname":"Ponting","address":{"city":"San Pedro","state":"LA"}}]

**Method --> DELETE,**

1. url -->http://localhost:8080/employee/2
2. Response :
3. [{"id":"1","firstname":"James","lastname":"Johnson","address":{"city":"Hoseynabad","state":"Kavir"}},
4. {"id":"3","firstname":"Roger","lastname":"Ponting","address":{"city":"San Pedro","state":"LA"}}]

# Go JSON

Go has built-in support for JSON encoding and decoding. it also supports custom datatypes.

The Marshal function is used to convert go data types into JSON format.

Marshal function syntax is:

1. "func Marshal(v interface{}) ([]byte, error)"

Marshal returns the JSON encoding of v.

Boolean is converted to JSON booleans. Floating point, integer, and Number are converted to JSON numbers. The map's key type must either be a string, an integer type, or implement encoding.TextMarshaler.

The decoding of JSON is done using Unmarshal function.

Unmarshal function syntax is:

1. "func Unmarshal(data []byte, v interface{}) error"

Unmarshal decodes JSON-encoded value and stores the result in the value pointed to by v. If v is nil or not a pointer, Unmarshal returns an InvalidUnmarshalError.

We can also customize the fields stored under the "json" key in the struct field's tag. we can have name of the field, followed by a comma-separated list of options. Like

1. Field **int** 'json:"myName"' // The appears in JSON as key "myName".
2. Field **int** 'json:"myName,omitempty?'// The field is omitted from the object if its value is empty,
3. Field **int** 'json:"-"' //// Field is ignored by this package.

### **Go JSON Example 1**

**package** main

**import** "encoding/json"

**import** "fmt"

func main() {

    bolType, \_ := json.Marshal(**false**) //boolean Value

    fmt.Println(string(bolType))

    intType, \_ := json.Marshal(10) // integer value

    fmt.Println(string(intType))

    fltType, \_ := json.Marshal(3.14) //float value

    fmt.Println(string(fltType))

    strType, \_ := json.Marshal("JavaTpoint") // string value

    fmt.Println(string(strType))

    slcA := []string{"sun", "moon", "star"} //slice value

    slcB, \_ := json.Marshal(slcA)

    fmt.Println(string(slcB))

    mapA := map[string]**int**{"sun": 1, "moon": 2} //map value

    mapB, \_ := json.Marshal(mapA)

    fmt.Println(string(mapB))

}

Output:

false

10

3.14

"JavaTpoint"

["sun","moon","star"]

{"moon":2,"sun":1}

### **Go JSON Example 2 (User Defined Data Type)**

**package** main

**import** (

    "encoding/json"

    "fmt"

    "os"

)

type Response1 struct {

    Position   **int**

    Planet []string

}

type Response2 struct {

    Position   **int**      'json:"position"'

    Planet []string 'json:"planet"'

}

func main()  {

    res1A := &Response1{

        Position:   1,

        Planet: []string{"mercury", "venus", "earth"}}

    res1B, \_ := json.Marshal(res1A)

    fmt.Println(string(res1B))

    res2D := &Response2{

        Position:   1,

        Planet: []string{"mercury", "venus", "earth"}}

    res2B, \_ := json.Marshal(res2D)

    fmt.Println(string(res2B))

    byt := []**byte**('{"pi":6.13,"place":["New York","New Delhi"]}`)

    var dat map[string]**interface**{}

**if** err := json.Unmarshal(byt, &dat); err != nil {

        panic(err)

    }

    fmt.Println(dat)

    num := dat["pi"].(float64)

    fmt.Println(num)

    strs := dat["place"].([]**interface**{})

    str1 := strs[0].(string)

    fmt.Println(str1)

    str := `{"Position": 1, "Planet": ["mercury", "venus"]}`

    res := Response2{}

    json.Unmarshal([]**byte**(str), &res)

    fmt.Println(res)

    fmt.Println(res.Planet[1])

    enc := json.NewEncoder(os.Stdout)

    d := map[string]string{"1":"mercury" , "2": "venus"}

    enc.Encode(d)

}

Output:

{"Position":1,"Planet":["mercury","venus","earth"]}

{"position":1,"planet":["mercury","venus","earth"]}

map[pi:6.13 place:[New York New Delhi]]

6.13

New York

{1 [mercury venus]}

venus

{"1":"mercury","2":"venus"}