

# Roadmap Reference

Here's some roadmap reference to learn <https://roadmap.sh/pdfs/go.pdf>

---

## Go

Why Use Go ? There's Node JS

It is a compiled driven language, progress in the directory according to the <https://www.openmymind.net/assets/go/go.pdf>  
we can develop the application we want using this dir

```
mkdir {{parent_dir}}/go/src
```

Using the go programming language we can try to create the main function.  
Using the main package example code

```
package main

func main() {
    println("Helo World")
}
```

After we write the code we can compile with

```
go run main.go
```

or if you want to create the executable files you can create the go build main.go.

```
go build main.go
```

then we can have the executable of the source code of hello world program.

That's basically the process so we can try to write the Makefile.

---

## Running every Section

To run every section of this writing excercices you can try to type this.

```
make build run
```

Generally I write the make file just like below snippets

```
build:
    cd ./bin/; go build ../src/main.go;

run:
    cd ./bin; ./main; cd ..
```

so the directory of every exercises should be

```
.
└─ {{Exercices Name}}
   └─ go
      ├── bin
      │   └─ main
      ├── Makefile
      └─ src
          └─ main.go
```

That way we can try to run the compiled program using Makefile.

## Running locally documentation

documentation at

```
godoc -http=:6060
```

## Go Modules

It's an additional features into which to simplify the package for Go Workspace Program.

## Go Workspace

Go Workspace is opinionated for efficiency and organized project with the team. There's certain convention in the go workspace. Example for using the Folder

```
.
└─ bin -----> for binary files
   └─ main -----> the binary files
└─ Makefile -----> where to create the makefile
└─ pkg -----> using the package (archived files)
└─ src -----> source code
   └─ github.com -----> the control integration provider maybe github or
gitlab
   └─ <username> -----> the username in that provider
      └─ <folder with the code for project or repo> ----> the username in
that provider
```

The pkg folder is to put the file of compiled archived library so we don't need to recompile.

## Go Help Command

```
go help
```

## Environment Variables

Two Important Environment Variables

```
GOPATH = workspace  
GOROOT = binary installation of Go
```

Just Type

```
go env
```

## Create the module.

First of all we need to create the module in the project directory with this command

```
go mod init <name of the module>
```

## Auto Formatting the Files

So Go have this features with the command `fmt` so all code will look the same. Just run

```
go fmt <dir/files you want to format>
```

So for examples if we create the files `src/main.go` like this

```
package main  
  
import ( "fmt" )  
  
func main() {  
    fmt.Println("Hello World") --> this is bad  
}
```

It can get some troubles if we work we a team so we can go

```
go fmt src/main.go
```

It will automatically format the code for us just like below snippets.

```
package main  
  
import ( "fmt" )  
  
func main() {  
    fmt.Println("Hello World") --> this is better  
}
```

## Go Modules

Go Modules is to managing the packages we install, so if our software is dependant about a software we should just configure it with the go modules.

ref: <https://go.dev/blog/using-go-modules>

OCT 15th WISUDA KA OKA.

## Initialize

creates a new module

```
go mod init
```

We can create the file dir for the module:

```
mkdir new-exercises
```

then after creating the directory we go into that directory.

```
cd new-exercises
```

then launch into programming code editor

```
touch hello.go && nvim hello.go
```

and after that module we can put the code in the `hello.go`

```
package hello

func Hello() string {
    return "Hello, World"
}
```

and create more such as `hello_test.go`

```
package hello

import "testing"

func TestHello(t *testing.T) {
    want := "Hello, World"
    if got := Hello(); got != want {
        t.Error
    }
}
```

when creating the go-modules we should create the namespacing such as `bin/` `src/`

such as this below.

```
go mod init new-exercises.com/gricowijaya/new-exercises
```

after that we can get the `go.mod` file.

we can try to cat the `go.mod` file

## Adding the dependency

```
go get github.com/rsc/quote
```

then do testing the module is running or not

```
go test
```

then after it's all good we can modify the file `hello.go` file

in the `hello` file is we can create the import file and after that we can do `go mod init`

```
package hello

import "rsc.io/quote"

func Hello() string {
    return "Hello, World"
}
```

There are direct dependencies and indirect dependencies. --> search more about this.

If we want to get all package list then we can create

```
go list -m all
```

To get all the packages

NOTES : both `go.mod` and `go.sum` must be checked in the version control

## Upgrading the dependency

We should list the file first such as using this command:

```
go list -m all
```

after that we can also get the try to get the content of the `go.mod` file :

```
cat go.mod
```

for example we want to upgrade the `rsc.io/sampler`

just type

to get the latest

```
go get rsc.io/sampler
```

to specify the version

```
go get rsc.io/sampler@v0.3.0
```

## Packages

To Use the standard library you can use this link to read the documentation

<https://pkg.go.dev/std>

`_` (throwing null error)

every program have package main and func main();

for example:

```
package main

import ( "fmt" )

func main() {
    n, _ := fmt.Println("Hello, world", 42, true)
    fmt.Println(n)

    // the _ is returning null error (if there's an error);
}
```

## Short Declaration operator

example in the short-declaration-operator/ directory

The Short Declaration Variables is allowing to write code and

example

```
x := 42 // declare and assign

fmt.Println(x)

y := 100 + 24 // making the expression for a statement

fmt.Println(y)
```

## Identifiers

Identifiers --> is the program entities such as variables and types.

An identifier is a sequence of one or more letters and digits.

It must be a letter;

There predeclared identifiers

## The var Keyword

The var keyword for var is just to get the value for shortage for variables.

There's also a zero value.

```
package main

import ("fmt")

// declare and assign
var y = 49

func main() {
    x := 42
    fmt.Println
}
```

## Data Type

To get the type we can use the

```
fmt.Printf("%T", y);
```

The Data type in Go is static not dynamic like Javascript.

It Has Primitive and Composite Data types (Array, Strings etc).

## Zero Value

Declare a variable to be a certain type.

```
package main

import ("fmt")

var y string

func main() {
    fmt.Println("start", y, "ending"); // output : start y ending
}

// assign into zero value
y = "Chaca"
```

## Fmt Package

The Format package or fmt has many methods such as Print, Println, Printf, Scanf

To take input:

```
package main

import ("fmt")

var y int;

func main() {
    fmt.Scanf("%y"); // stdin
    fmt.Println(y); // stdout
}
```

## Converting type

In the go programming language is kinda like the casting feature in C but in Go it's called converting which used for changing data types to Type

```
type Integer int

import ("fmt")

func main() {
    var i int;
    i = 1;
    fmt.Printf("%T",i); // output :int
    i = Integer(i)
    fmt.Printf("%T",i); // output :string
}
```

## Details on The Numeric Types

In the Numeric Types we can try to create such as uint8 (unsigned integer 8-bit) int8 etc

<https://www.geeksforgeeks.org/data-types-in-go/>

## String

In Go Programming Language String is just a set of empty sequences of bytes. The predeclared string type is called string there are a slice of bytes.



```

package main

import("fmt")

func main() {
    s := "Hello World"
    fmt.Println(s);
    fmt.Println("%T\n", s);

    bytesString := []byte(s); // print the slice of bytes
    fmt.Println(bytesString);
    fmt.Println("%T\n", bytesString);
}

```

## Constants

Which is the keyword const we can try to implement the const in this snippets

```

package main

import("fmt")

const a = 1;
const b = 2;
const c = 3;

func main() {
    fmt.Println(a);
    fmt.Println(b);
    fmt.Println(c);
}

```

Another way to declare this is just like below.

```

package main

import("fmt")

const (
    a int = 1
    b float64 = 2
    c string = 3
)

```

## Iota

Is a special character that can be used as an auto  
it can be written such as using the code below

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

## Bit Shifting

We can transform the bit number into int with some of this technique.  
Using Iota let's try to create it !

```
package main
```

Why we should learn about the bit shifting ? Let's take an example here:

```
package main

import (
    "fmt"
)

func main() {

    kb := 1024
    mb := 1024 * kb
    gb := 1024 * mb

    fmt.Println("\t[decimal]\t\t[binary]");
    fmt.Printf("[kb] == %d\t\t\t%b\n", kb, kb); // output :
    fmt.Printf("[mb] == %d\t\t\t%b\n", mb, mb); // output :
    fmt.Printf("[gb] == %d\t\t\t%b\n", gb, gb); // output :
}
```

The output from the program above is like the below snippets

[illegible]

that is with a declarable variable with `iota` we can manipulate the program so it'll be more automated declaration by the go compiler, because the `kb` `mb` `gb` is incremented by default with `iota` declaration for example of code we can write just like the below code.

```
package main

import (
    "fmt"
)

// built with iota
const (
    _ = iota;
    kb = 1 << ( iota * 10 )
    mb = 1 << ( iota * 10 )
    gb = 1 << ( iota * 10 )
)

func main() {
    fmt.Println("\t[decimal]\t\t[binary]");
    fmt.Printf("[kb] == %d\t\t%b\n", kb, kb);
    fmt.Printf("[mb] == %d\t\t%b\n", mb, mb);
    fmt.Printf("[gb] == %d\t\t%b\n", gb, gb);
}
```

it will create the same output from the program before but as we can see it is a good practice by using iota in shifting an incremental values rather than declare a variables, but it is a personal preferences.

For more Bit Hacking we can also reference to this article :

<https://medium.com/learning-the-go-programming-language/bit-hacking-with-go-e0acee258827>

## Exercises

In Go we can write the assign a comparison between a value such as <= >= etc. The code is just like below.

```
package main

import ("fmt")

func main() {
    a := ( 42 == 42) // output : true
    b := ( 42 <= 43) // output : true
    c := ( 42 >= 43) // output : false
    d := ( 42 != 43) // output : true
    e := ( 42 < 43) // output : true
    f := ( 42 > 43) // output : false

    fmt.Println(a, b, c, d, e, f);
}
```

In the code above we can assign the value of a b c d e f for getting the boolean value of the condition is true or false

## TYPED and UNTYPED Constants

We can expand the program above using the TYPED and UNTYPED values. the Procedure we need to create is just like this.

1. We need to set the constant variable first.
2. Assign the variable into using the typed and untyped variabls

```

package main

import("fmt")

const (
    a = 42 // UNTYPED constant -- flexibility for the compiler beware they can
    read this as a string.
    b int = 43 // TYPED constant -- more precise type for the compiler.
)

func main() {
    fmt.Println(a, b);
    fmt.Printf("%T, %T", a, b); // if we want to see the data type of the
    variables
}

```

## Control Flow in Go

In Go Programming Language we can use the control flow to create the such loop

A Loop contain a init, condition, and post. You may ask, what are those ?

init stands for initialization variables, condition is for conditioning the variables in the looping section such as how to do want control this looping (because of course we don't want a infinite looping program) and post is control is to update the condition so the init also will be updated.

### For Loop

```

package main

import("fmt")

func main() {
    for i := 1; i < 4; i++ {
        fmt.Printf("loop iteration of %d\n", i );
    }
}

```

The output for the program above is will be like

```

loop iteration of 1
loop iteration of 2
loop iteration of 3

```

Well the above program is a simple program which will run just word of 'loop iteration of {number of iteration}'

Meanwhile go there are so many ways to write iteration.

In the go environment we can use many keyword for using iteration

## Switch Case

Create the switch case in the go programming language there's a feature of it which is called fallthrough.

fallthrough is just like when the condition is true then told the program to continue.

The value of switch expression

```
package main

import("fmt");

func main(){
    n := 2;
    switch nx {
        case 1:
            fmt.Println("Case 1");
            fallthrough
        case 2:
            fmt.Println("Case 2");
            fallthrough
        default:
    }
}
```

## Usage of break and continue

The usage of break and continue in the go programming is that they can be used as a control condition for loops

For example

```
package main

import("fmt")

func main() {
    for i := 1; i < 3 ; i++ {
        if i == 3 {
            break;
        }

        fmt.Println("it's already broken in ", i);
    }
}
```

## Array

Declaring an array in go programming language can be written in different method such as

in programming is an 0 based index

```
package main

import("fmt")

func main() {
    array := [3]string{"one", "two", "three"}
    fmt.Print(array); // output [one two three]
}
```

We also can use the var keyword just like the below snippets

```
package main

import("fmt")

func main() {
    var number_of_array = [3]int{1, 2, 3}
    fmt.Println(number_of_array[1]) // output = 2
}
```

for accessing the array we can use the loops such as for in range or regular for loops just like the snippets below:

```
package main

import("fmt")

func main() {
    var number_of_array = [10]one{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    for i := 1; i <= 10 ; i++ {
        fmt.Println(`using the for loop`, array[i]);
    }

    for _, value := range number_of_array { // use a blank identifiers
        fmt.Println(`using the for in range loop `, value);
    }
}
```

you can choose however you like to loops the array either with the regular for loops or the for in range loop

Arrays are useful but use slices instead.

one more example

```

package main

import("fmt")

func main() {
    var x[3]int
    var y[10]int
    fmt.Printf("%t", x)
    fmt.Printf("%t", y.len)
}

```

To get a length of an array we can use the len function.

## Slices

Slices is one of the Aggregate data types which sometime called structure types which is composite data types which called structure.

Which is a composite literal.

```

package main

import("fmt")

func main() {
    // x := []type{values} COMPOSITE literal
    x := []int{1 , 2, 3, 4, 5, 6} // x is a slice ([]) of int
    fmt.Println(x);
}

```

To loop over a value in a slice we can try to create just like the below functions

```

package main

import("fmt")

func main() {
    //
    x := []int{1 , 2, 3, 4, 5, 6} // x is a slice ([]) of int
    fmt.Println(x);

    for index, value := range x {
        fmt.Printf("This is the element of %d is %d", index, value)
    }
}

```

After we try to create the slices of slice using the : (colon operator)



```

package main

import("fmt")

func main() {
    //
    x := []int{10, 20, 30, 40, 50, 60} // x is a slice ([]) of int
    fmt.Println(x[1:2]); // output will be 20 because will get the element that
    start from index 1 up to element that before index 2 which is 20
}

```

In Slice we also can append to a slice which we can add the value into it.

```

package main

import("fmt")

func main() {
    x := []int{10, 20, 30, 40, 50, 60} // x is a slice ([]) of int
    fmt.Println("before append", x)
    x = append(x, 80, 70, 90 )          // append 80, 70, 90 into x
    fmt.Println("after append", x)

    y := []int{1,45,433,432,678}
    x = append(x, y...)

    fmt.Println("After appended with y ", y)

    x = append(x[:5], x[6:]...) // remove the element between the 5th and 6th
index
    fmt.Println("After remove the element ", x)
}

```

## Multi Dimensional Slice

We can declare the multi dimensional slice just like the main slice declaration for the go programming language except with a slices in side the slices for example:

```

package main

import("fmt")

func main() {
    x := []int{10, 20, 30, 40, 50, 60} // x is a slice ([]) of int
    y := []int{1,45,433,432,678}

    z := [][]int{x, y}
    fmt.Println("multi dimensional slices ", z) // output will be [[10, 20, 30,
40, 50, 60] [1, 45, 433, 432, 678]]
}

```

think of it just like a spreadsheet to better visualization

```
[
  [10, 20, 30, 40, 50, 60],
  [1, 45, 433, 432, 678]
]
```

## Map

What is maps is a key value store which is store some value based on the key.

Language specification

```
package main

import("fmt")

func main() {
  m := map[string]int{
    "James" : 32,
    "Miss Money": 28,
  }

  fmt.Println(n);
  fmt.Println(m["James"]) // output : 32
  fmt.Println(m["Barnabas"]) // output : 0 -> because the value is not stored
in the maps

  v, ok := m["Barnabas"] // use the comma ok idiom
  fmt.Println(m["Barnabas"]) // output : 0
  fmt.Println(v) // output : 0
  fmt.Println(ok) // output : false

  // we can combine it with an if statme just like this
  if v, ok := m["Barnabas"]; ok {
    fmt.Println("It's in there")
  }
}
```

## Map

Hold references to an underlying data structure it can be distinguish a missing entry of value.

```

package main

import("fmt")

func main() {
    m := map[string]int{
        "James":32,
        "Miss Money Penny":27,
    }
    fmt.Println(m)
    fmt.Println(m["James"])
    fmt.Println(m["Barnabas"])

    v, ok := m["Barnabas"]
    fmt.Println(v)
    fmt.Println(ok)

    if v, ok := m["Barnabas"]; ok { fmt.Println(v) }
}

```

## Adding Element & Range

```

package main

import("fmt")

func main() {
    m := map[string]int{
        "James":32,
        "Miss Money Penny":27,
    }
    fmt.Println(m)
    fmt.Println(m["James"])
    fmt.Println(m["Barnabas"])

    v, ok := m["Barnabas"]
    fmt.Println(v)
    fmt.Println(ok)

    m["Todd"] = 44
    if v, ok := m["Barnabas"]; ok { fmt.Println(v) }

    for k, v := range m {
        fmt.Println("Print the ", k, v);
    }

    xi := []int{4,2,3,4,1,54}

    for i, v := range xi {
        fmt.Println(i, v);
    }
}

```

## Delete from a Map

```
package main

import ("fmt")

func main() {
    m := map[string]int{
        "James":32,
        "Miss Moneypenny":27,
    }
    fmt.Println(m)
    delete(m, "James")

    delete(m, "Ian Fleming")

    fmt.Println(m["Miss Moneypenny"])
    fmt.Println(m["Ian Fleming"])

    if v, ok := m["Miss Moneypenny"]; ok {
        fmt.Println("value", v);
        delete(m, "Miss moneypennny")
    }
}
```

## Structs

Struct is a data structure that consists of different Data Type which can be aggregated

```
package main
import("fmt")

type person struct {
    first string
    last  string
}

func main() {
    p1 := person {
        first: "James",
        last: "Bond",
    }

    p2 := person {
        first: "Miss"
        first: "Money Penny"
    }

    fmt.Println(p1)
    fmt.Println(p2)
}
```

## Embedding Type

To approaching the OOP in Go we can try to create just like this.

```
package main
import("fmt")

type person struct {
    first string
    last  string
}

type secretAgent struct {
    person
    ltk bool // license to kill
}

func main() {

    sa1 := secretAgent{
        person: person {
            first: "James",
            last:  "Bond",
        },
        ltk: true,
    }

    p2 := person {
        first: "Miss"
        first: "Money Penny"
    }

    fmt.Println(p1)
    fmt.Println(p2)
}
```

## Anonymous Structs

If We need a struct in little area.

```
import("fmt")

type person struct {
}

func main() {

    p1 := struct {
        first string
        last  string
        age   int
    }{
        first: "James",
        last:  "Name",
        age:   32 ,
    }

    fmt.Println(p1)
}
```

## Functions

We Can use packages or functions which we can use to programming.

The Function parameters

```
func (r receiver) identifiers(parameters) (return(s)) {...}
```

```
func foo() {
    fmt.Println("Hello From Foo")
}
```

We can pass parameter using the pass by value for example

```
func name(s string) {
    fmt.Println("Hello my name is ", s)
}
```

To return a value we can create how to use the

```
func woo()
```

## Variadic Parameters

```
package main

import "fmt"

func main() {
    foo(1, 2, 3);
}

// foo is a variadic function.
// It can be called in the following ways:
// foo()
func foo(args ...int) {
    fmt.Println(args)
}
```

---

## Methods

---