# Module 4

## Package in Go language :

- Packages are the most powerful part of the Go language.
- The purpose of a package is to design and maintain a large number of programs by grouping related features together into single units so that they can be easy to maintain and understand and independent of the other package programs.
- This modularity allows them to share and reuse.
- In Go language, every package is defined with a different name and that name is close to their functionality like “strings” package and it contains methods and functions that only related to strings.
- Every Go source file belongs to a package. To declare a source file to be part of a package, we use the following syntax -

```
package <packagename>
```

Every Go program starts with the main package. Whenever the compiler sees the main package, it treats the program as the executable code.

**Import paths:**

In Go language, every package is defined by a unique string and this string is known as import path. With the help of an import path, you can import packages in your program. For example:

```
import "fmt"
```

This statement states that you are importing an fmt package in your program. The import path of packages is globally unique. To avoid conflict between the path of the packages other than the

standard library, the package path should start with the internet domain name of the organization that owns or host the package. For example:

```
import "geeksforgeeks.com/example/strings"
```

**Package Declaration:**

In Go language, package declaration is always present at the beginning of the source file and the purpose of this declaration is to determine the default identifier for that package when it is imported by another package. For example:

```
package main
```

**Import declaration:**

The import declaration immediately comes after the package declaration. The Go source file contains zero or more import declaration and each import declaration specifies the path of one or more packages in the parentheses. For example:

```
// Importing single package
import "fmt"

// Importing multiple packages
import(
"fmt"
"strings"
"bytes"
)
```

When you import a package in your program you're allowed to access the members of that package. For example, we have a package named as a "sort", so when you import this package in your program you are allowed to access sort.Float64s(), sort.SearchStrings(), etc functions of that package.

**Blank import:**

In Go programming, sometimes we import some packages in our program, but we do not use them in our program. When you run such types of programs that contain unused packages,

then the compiler will give an error. So, to avoid this error, we use a blank identifier before the name of the package. For example:

```
import _ "strings"
```

It is known as blank import. It is used in many or some occasions when the main program can enable the optional features provided by the blank importing additional packages at the compile-time.

**Nested Packages:**

In Go language, you are allowed to create a package inside another package simply by creating a subdirectory. And the nested package can import just like the root package. For example:

```
import "math/cmplx"
```

Here, the math package is the main package and cmplx package is the nested package.

Sometimes some packages may have the same names, but the path of such type of packages is always different. For example, both math and crypto packages contain a rand named package, but the path of this package is different, i.e, math/rand and crypto/rand.

In Go programming, why always the main package is present on the top of the program? Because the main package tells the go build that it must activate the linker to make an executable file.

## Giving Names to the Packages:

In Go language, when you name a package you must always follow the following points:

- When you create a package the name of the package must be short and simple. For example strings, time, flag, etc. are standard library package.
- The package name should be descriptive and unambiguous.
- Always try to avoid choosing names that are commonly used or used for local relative variables.

- The name of the package generally in the singular form. Sometimes some packages named in plural form like strings, bytes, buffers, etc. Because to avoid conflicts with the keywords.
- Always avoid package names that already have other connotations. For example:

**Example:**

```
// Go program to illustrate the
// concept of packages
// Package declaration
package main

// Importing multiple packages
import (
   "bytes"
   "fmt"
   "sort"
)

func main() {

   // Creating and initializing slice
   // Using shorthand declaration
   slice_1 := []byte{'*', 'G', 'o', 'P', 'r', 'o', 'g',
      'r', 'a', 'm', 'm', 'i', 'n', 'g', '^', '^'}
   slice_2 := []string{"Go", "Gopher", "format", "variables", "structs"}

   // Displaying slices
   fmt.Println("Original Slice:")
   fmt.Printf("Slice 1 : %s", slice_1)
   fmt.Println("\nSlice 2: ", slice_2)

   // Trimming specified leading
   // and trailing Unicode points
   // from the given slice of bytes
   // Using Trim function
   res := bytes.Trim(slice_1, "*^")
   fmt.Printf("\nNew Slice : %s", res)
```

```
    // Sorting slice 2
    // Using Strings function
    sort.Strings(slice_2)
    fmt.Println("\nSorted slice:", slice_2)
}
```

**Output:**

Original Slice:

Slice 1 : *GoProgramming^^

Slice 2:  [Go Gopher format variables structs]

New Slice : GoProgramming
Sorted slice: [Go Gopher format structs variables]

## Reflection Package:

Reflection in Go is a form of metaprogramming. Reflection allows us to examine types at runtime. It also provides the ability to examine, modify, and create variables, functions, and structs at runtime. The Go reflect pacakge gives you features to inspect and manipulate an object at runtime. Reflection is an extremely powerful tool for developers and extends the horizon of any programming language. Types, Kinds and Values are three important pieces of reflection that are used in order to find out information.

### The functions of  Reflection Package:

1. reflect.Copy()
2. reflect.DeepEqual()
3. reflect.Swapper()
4. reflect.TypeOf()
5. reflect.ValueOf()
6. reflect.NumField()
7. reflect.Field()

8. reflect.FieldByIndex()
9. reflect.FieldByName()
10. reflect.MakeSlice()
11. reflect.MakeMap()
12. reflect.MakeChan()
13. reflect.MakeFunc()

## reflect.Copy()

Copy copies the contents of source into destination until either destination has been filled or source has been exhausted. It returns the number of elements copied. Destination and source each must have kind Slice or Array, and destination and source must have the same element type.

### Example:

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

func main() {
   destination := reflect.ValueOf([]string{"A", "B", "C"})
   source := reflect.ValueOf([]string{"G", "O", "P", "L"})

   // Copy() function is used and it returns the number of elements copied
   counter := reflect.Copy(destination, source)
   fmt.Println(counter)

   fmt.Println(source)
   fmt.Println(destination)
}
```

Output:

3

[G O P L]

[G O P]

## reflect.TypeOf()

The reflect.TypeOf function returns a value of type reflect.Type, which represents the type of the variable passed into the TypeOf function.

Example:

```
package main

import (
   "fmt"
   "reflect"
)

func main() {
   v1 := []int{1, 2, 3, 4, 5}
   fmt.Println(reflect.TypeOf(v1))

   v2 := "Hello World"
   fmt.Println(reflect.TypeOf(v2))

   v3 := 1000
   fmt.Println(reflect.TypeOf(v3))

   v4 := map[string]int{"mobile": 10, "laptop": 5}
   fmt.Println(reflect.TypeOf(v4))

   v5 := [5]int{1, 2, 3, 4, 5}
   fmt.Println(reflect.TypeOf(v5))
```

```
    v6 := true
    fmt.Println(reflect.TypeOf(v6))
}
```

Output:

[]int

string

int

map[string]int

[5]int

Bool

## reflect.MakeMap()

MakeMap creates a new map with the specified type.

Example:

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var str map[string]string
    var strType reflect.Value = reflect.ValueOf(&str)
    newMap := reflect.MakeMap(reflect.Indirect(strType).Type())

    fmt.Println("Go Programming :", newMap.Kind())
}
```

**Output:**

Go Programming : map

## reflect.MakeChan()

MakeChan creates a new channel with the specified type and buffer size.

**Example:**

```
package main

import (
        "fmt"
        "reflect"
)

func main() {
        var str chan string
        var strType reflect.Value = reflect.ValueOf(&str)
        newChannel := reflect.MakeChan(reflect.Indirect(strType).Type(), 512)

        fmt.Println("Kind:", newChannel.Kind())
        fmt.Println("Capacity :", newChannel.Cap())
}
```

**Output:**

Kind : chan

Capacity: 512

## reflect.MakeFunc()

The reflect.MakeFunc() Function is used to get the new function of the given Type that wraps the function fn.

**Example:**

```
package main

import (
   "fmt"
   "reflect"
)

type Sum func(int64, int64) int64

func main() {
   t := reflect.TypeOf(Sum(nil))
   mul := reflect.MakeFunc(t, func(args []reflect.Value) []reflect.Value {
      a := args[0].Int()
      b := args[1].Int()
      return []reflect.Value{reflect.ValueOf(a + b)}
   })
   fn, ok := mul.Interface().(Sum)
   if !ok {
      return
   }
   fmt.Println(fn(7, 6))
}
```

## Output:

13

## Tools for the Go language :

GoLang tools play an essential role when it comes to design a web application. It helps a lot in the coding section of the project.

1. Go Vendor
2. Gonative
3. Depth
4. Checkstyle
5. Apicompat
6. Go-Swagger
7. Go-Callvis
8. Go Meta Liner
9. Go Simple
10. Grapes Tool

**Depth :**

This tool is meant to facilitate the web developers in retrieving and visualizing the source code dependency trees in the go language. The depth tool can be used in your project as a particular package. You can also use this tool as a stand-alone command-line application. It allows you to add customization very easily to your program.

**Checkstyle :**

This is another important tool of the go language. The go language checkstyle tool is inspired by the java checkstyle and the go language's golint tool. In the go language, this tool prints out the coding style suggestions. It also allows the go developer to check the file line-function and file line-param number.

There are some alternatives to the checkstyle tool also. These alternatives tools are:

1. To import
2. Login
3. Go metaliner

**Apicompat :**

Apicompat is an important tool of the go language. This tool helps the developer in detecting the incompatible changes and backward in the go program. It also checks the exported declarations in the program. It avoids false positives.

**Go-Swagger :**

Go swagger is designed from the impression of swagger 2.0, and the go swagger can serialize and DE-serialize the functions of swagger 2.0. Go swagger tool provides the complete suite of GoLang feature to the developer. The tool brings the API components to work with swagger API, i.e., server, client, and date model.

Features :

1. It provides the code generation for the go program.
2. It also provides API generation based on swagger specifications.
3. It extends the string format.
4. It provides great customization of features in the go program.

**Go Meta Liner:**

This one is another important tool of the go language. You can use the go meta liner tool as a substitute for the go lint tool. If you are required to use the go-lint tool and normalize the output, you can simply do it with the help of the go meta liner tool.

You can use the go meta liner tool with a text editor or integrate with an IDE like you can use Sublime text with Sublime linter plug-in, Atom with a go-plus package, Vim/Neovim, and go for Visual Studio Code. It supports a large number of linters and their configuration files, the same as JSON.

**Go Simple :**

It is a very simple but important tool of the go language. The main feature of the go simple tool is that it simplifies the go source code on the priority level. In all its functionality, it mainly focuses on simplifying the source code of your go program.

The go simple tool always supports the latest version of the go language. If the new version of the go language releases and your current version is outdated, then the go simple tool will suggest to you the easiest and the simplest way to ignore the complicated constructions in your go program.

**Grapes Tool:**

The Grapes is a very light weighted tool of the go language. This tool is mainly designed for distributing the commands over SSH (secure shell, use to establish secured and encrypted communication between two hosts) without any difficulty. The Grapes tool of the go language is developed by Yaron Sumel and is currently maintained by him.

The Grapes tool is not gaining the developers' attention at the current time, and it is under the maintenance process for enhancing its functions and features.

**Gotest tool:**

The gotest is a tool that is used to generate the go tests in the program. It generates the go tests from the source code of your go program. It makes it easier for the developer to write the go tests.

The gotest is the go language's command-line tool, which is used in generating the table-driven tests based on functions and method signatures available on its target source file.

## The test functions in GoLang

Test functions are essential but it not always obvious how we can run tests in a better and efficient way. Writing the same code over and over again can take a huge amount of time. This post aims to provide some insights into what can be done to reduce this problem.

## Code Coverage:

Code coverage is the way to determine the test coverage of a package. Test coverage is a metric that provides an outline of how much of the functions are covered by tests.

**The rules to follow :**

To do test coverage in Go we create a test file by adding a _test suffix.

```
filename_test.go
```

Then we simply use test functions in that file. To generate the coverage report, the -cover flag is added after that.

```
go test -cover
```

This cover flag tells the test tool to generate coverage reports for the package we are testing.

**Profiling:**

Profiling is a form of analyzing the program for optimizable code or functions. In software engineering, it is an essential task since optimization is a key factor when developing an application. Avoiding memory leaks and optimizing for better performance is almost always a target for enterprise-level software.

## Benchmark:

A benchmark by the dictionary definition is a standard, with which the same type of things may be compared. In programming, benchmark tests the time of execution for an operation. For example, a complex function execution time or a simple function executed a million times can be considered for benchmarking.

**Required imports for Benchmarking in Go:**

The testing package has benchmark functions which can be used directly to produce benchmarks.

```
import "testing"
```

**Benchmark rules in Go :**

To do benchmark the functions defined for the benchmark must be of the form as shown below.

```
func BenchmarkXXX(b *tesing.B){
        // do benchmark...
}
```

When running the go test we must provide the -bench flag which will show the output.

```
go test -bench=.          // the dot is the regex matching everything
```