Unit – 4

Packages, The Core Packages - Strings, Input/Output, Files and Folders, Errors, Containers and Sort, Hashes and Cryptography, Servers - TCP, HTTP–RPC.

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.

1.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 "go.com/example/strings"

2. 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

3. 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

4. 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.

5. 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.

6. 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.

7. 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.

Commonly used packages in Go

Now that we know how to import packages, let's learn about some of the popular packages:

fmt Package

math Package

string Package

Golang fmt Package

In Go, the fmt package provides functions to format our input/output data. For example, the fmt.Println() function prints the data to the output screen.

Some of the commonly used fmt functions:

Functions	Descriptions
Print()	prints the text to output screen
Println()	prints the text to output with a new line character at the end
Printf()	prints the formatted string to the output screen
Scan()	get input values from the user
Scanf()	get input values using the format specifier
Scanln()	get input values until the new line is detected
Println()	prints the text to output with a new line character at the end

Example 1: Golang fmt package

```
package main
import "fmt"

func main() {
var number int
// take input value fmt.Scan(&number)

// print using Println
fmt.Println("Number is", number)

fmt.Print("Using Print")
fmt.Println("Using Println")

}
```

Output

```
Number is 10
Using PrintUsing Println
```

In the above example, we have used the fmt.Scan() function to take input value and assign it to the number variable. We then print the value of number using the fmt.Println().

The Println() function adds a newline character at the end by default. That's why the next statement, fmt.Print() prints the text, Using Print in the new line.

However, Print() doesn't add the newline character by default, the next print statement prints the text Using Println in the same line.

Example 2: fmt Scanf() and Printf() functions

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

var number int

 fmt.Scanf("%d", &number)  // Input: 10
 fmt.Printf("%d", number)  // Output: 10
}
```

In the above example, functions
fmt.Scanf("%d", &number) - takes integer input value and assign it to the number variable

fmt.Printf("%d", number) - replaces the %d format specifier by the value of number and prints it

math package in Go

The math package provides various functions to perform mathematical operations. For example, math.Sqrt() finds the square root of a number.

Some of the commonly used math functions:

Functions	Descriptions
Sqrt()	returns the square root of the number
Cbrt()	returns the cube root of the number
Max()	returns the larger number between two
Min()	returns the smaller number between two
Mod()	computes the remainder after division

To use these functions, we must import the math package.

Example: math Package

```
package main
import "fmt"

// import the math package
import "math"

func main() {

  // find the square root
  fmt.Println(math.Sqrt(25))    // 5

  // find the cube root
  fmt.Println(math.Cbrt(27))    // 3

  // find the maximum number
  fmt.Println(math.Max(21, 18))    // 21

  // find the minimum number
  fmt.Println(math.Min(21, 18))    // 18

  // find the remainder
  fmt.Println(math.Mod(5, 2))    // 1
}
```

Here, we have imported the math package in our program. This is why we are able to use math-based functions like Sqrt(), Max(), etc in our program.

Go strings package

The strings package provides functions to perform operations on UTF-8 encoded strings. For example, strings.Contains() checks if the string contains a substring.

Some of the commonly used strings functions:

Functions	Descriptions
Compare()	checks if two strings are equal
Contains()	checks if the string contains a substring

Count()	counts the number of times a substring present in the string
Join()	creates a new string by concatenating elements of a string array
ToLower()	converts the string to lowercase
ToUpper()	converts the string to uppercase

Example: string Package

```
package main

// import multiple packages
import (
  "fmt"
  "strings"
  )

func main() {

  // convert the string to lowercase

  lower := strings.ToLower("GOLANG STRINGS")fmt.Println(lower)
// convert the string to uppercase
 upper := strings.ToUpper("golang strings")
 fmt.Println(upper)

 // create a string array
 stringArray := []string{"I love", "Go Programming"}
joined string := strings.Join(stringArray,”);
fmt.Println(joinedString)
}
```

Output:

```
golang strings
GOLANG STRINGS
I love Go Programming
```

In the above example, we have used functions of the strings package to perform various operations on the strings.

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:

```
// Go program to illustrate the

// concept of packages

// Package declarationpackage main

// Importing multiple packagesimport (

"bytes""fmt"

"sort"

)

func main() {

// Creating and initializing slice

// Using shorthand declaration

slice_1 := []byte{'*', 'G', 'e', 'e', 'k', 's', 'f',

'o', 'r', 'G', 'e', 'e', 'k', 's', '^', '^'}

slice_2 := []string{"Gee", "ks", "for", "Gee", "ks"}

// 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 : *GeeksforGeeks^^
Slice 2: [Gee ks for Gee ks]


New Slice : GeeksforGeeks
Sorted slice: [Gee Gee for ks ks]
```

Go Custom Package

we have been using packages that are already defined inside the Go library. However, Go programming allows us to create our own custom packages and use them just like the predefined packages.

1. Create Custom Package

To create a custom package, we first need to create a new file and declare the package. For example,

```
// declare package

package calculator
```

Now, we can create functions inside the file. For example,

```
package calculator

// create add function func
Add(n1, n2 int) int {return
n1 + n2
}

// create subtract function func
Subtract(n1, n2 int) int {return
n1 - n2
}
```

In the above example, we have created a custom package named calculator. Inside the package,

we have defined two functions: Add() and Subtract().

2. Importing Custom Package

Now, we can import the custom package in our main file.

```
package main

// import the custom package calculator
import (
  "fmt" "Packages/calculator"
)
func main() {
  number1 := 9
  number2 := 5

  // use the add function of calculator package fmt.Println(calculator.Add(number1,
  number2))

  // use the subtract function of calculator package
  fmt.Println(calculator.Subtract(number1, number2))
  }
```

The Core Packages

Strings

Go includes a large number of functions to work with strings in the strings package:

```
package main

import ( "fmt" "strings")

func main() {
 fmt.Println(
// true strings.Contains("test", "es"),

// 2
strings.Count("test", "t"),

// true strings.HasPrefix("test", "te"),

// true strings.HasSuffix("test", "st"),

// 1
strings.Index("test", "e"),

// "a-b" strings.Join([]string{"a","b"}, "-"),
```

```
// == "aaaaa" strings.Repeat("a", 5),

// "bbaa"
strings.Replace("aaaa", "a", "b", 2),

// []string{"a","b","c","d","e"}
strings.Split("a-b-c-d-e", "-"),
      // "test"
      strings.ToLower("TEST"),

      // "TEST"
      strings.ToUpper("test"),
    )
  }
```

Input / Output

Before we look at files we need to understand Go's io package. The io package consists of a few functions, but mostly interfaces used in other packages. The two main interfaces are Reader and Writer. Readers support reading via the Read method. Writers support writing via the Write method. Many functions in Go take Readers or Writers as arguments. For example the io package has a Copy function which copies data from a Reader to a Writer:

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

To read or write to a []byte or a string you can use the Buffer struct found in the bytes package:

```
var buf bytes.Buffer

buf.Write([]byte("test"))
```

A Buffer doesn't have to be initialized and supports both the Reader and Writer interfaces. You can convert it into a []byte by calling buf.Bytes(). If you only need to read from a string you can also use the strings.NewReader function which is more efficient than using a buffer.

Files & Folders

To open a file in Go use the Open function from the os package. Here is an example of how to read the contents of a file and display them on the terminal:

```
package main

import ( "fmt"
"os"
)

func main() {
file, err := os.Open("test.txt") if err != nil {
// handle the error here
    return
   }
```

```
  defer file.Close()

  // get the file size stat,
  err := file.Stat()if err !=
  nil {
    return
  }
  // read the file
  bs := make([]byte, stat.Size())
  _, err = file.Read(bs)if
  err != nil {
    return
  }

  str := string(bs)
  fmt.Println(str)
}
```

Errors

Go has a built-in type for errors that we have already seen (the error type). We can create our own errors by using the New function in the errors package:

```
package main

import "errors"

func main() {
  err := errors.New("error message")
}
```

Containers & Sort

List

The container/list package implements a doubly-linked list. A linked list is a type of data structure that looks like this:

```
package main

import ("fmt" ; "container/list")

func main() {
  var x list.List
  x.PushBack(1)
  x.PushBack(2)
  x.PushBack(3)

  for e := x.Front(); e != nil; e=e.Next() {
    fmt.Println(e.Value.(int))
  }
}
```

The zero value for a List is an empty list (a *List can also be created using list.New). Values are appended to the list using PushBack. We loop over each item in the list by getting the first element, and following all the links until we reach nil

Sort

The sort package contains functions for sorting arbitrary data. There are several predefined sorting functions (for slices of ints and floats) Here's an example for how to sort your own data:

```
package main

import ("fmt" ; "sort")

type Person struct {
  Name string
  Age int
}

type ByName []Person

func (this ByName) Len() int {
  return len(this)
}
func (this ByName) Less(i, j int) bool {
  return this[i].Name < this[j].Name
}
func (this ByName) Swap(i, j int) {
  this[i], this[j] = this[j], this[i]
}

func main() {
  kids := []Person{
    {"Jill",9},
    {"Jack",10},
  }
  sort.Sort(ByName(kids))
  fmt.Println(kids)
}
```

The Sort function in sort takes a sort.Interface and sorts it. The sort.Interface requires 3 methods: Len, Less and Swap. To define our own sort we create a new type (ByName) and make it equivalent to a slice of what we want to sort. We then define the 3 methods.

Sorting our list of people is then as easy as casting the list into our new type. We could also sort by age by doing this:

```
type ByAge []Person
func (this ByAge) Len() int {
  return len(this)
}
```

```
func (this ByAge) Less(i, j int) bool {
  return this[i].Age < this[j].Age
}
func (this ByAge) Swap(i, j int) {
  this[i], this[j] = this[j], this[i]
}
```

Hashes & Cryptography

A hash function takes a set of data and reduces it to a smaller fixed size. Hashes are frequently used in programming for everything from looking up data to easily detecting changes. Hash functions in Go are broken into two categories: cryptographic and non-cryptographic.

The non-cryptographic hash functions can be found underneath the hash package and include adler32, crc32, crc64 and fnv. Here's an example using crc32

```
package main

import  ( "fmt"
  "hash/crc32"
)

func main() {
  h := crc32.NewIEEE()
  h.Write([]byte("test"))v
  := h.Sum32()
  fmt.Println(v)
}
```

The crc32 hash object implements the Writer interface, so we can write bytes to it like any other Writer. Once we've written everything we want we call Sum32() to return a uint32. A common use for crc32 is to compare two files. If the Sum32 value for both files is the same, it's highly likely (though not 100% certain) that the files are the same. If the values are different then the files are definitely not the same:

```
package main

import  ( "fmt"
  "hash/crc32"
  "io/ioutil"
)

func getHash(filename string) (uint32, error) {bs,
  err := ioutil.ReadFile(filename)
  if err != nil {
    return 0, err
  }
  h := crc32.NewIEEE()
  h.Write(bs)
  return h.Sum32(), nil
}
```

```
func main() {
  h1, err := getHash("test1.txt")if
  err != nil {
    return
  }
  h2, err := getHash("test2.txt")if
  err != nil {
    return
  }
  fmt.Println(h1, h2, h1 == h2)
}
```

Cryptographic hash functions are similar to their non-cryptographic counterparts, but they have the added property of being hard to reverse. Given the cryptographic hash of a set of data, it's extremely difficult to determine what made the hash. These hashes are often used in security applications.

One common cryptographic hash function is known as SHA-1. Here's how it is used:

```
package main

import (
  "fmt"
  "crypto/sha1"
)

func main() {
  h := sha1.New()
  h.Write([]byte("test"))
  bs := h.Sum([]byte{})
  fmt.Println(bs)
}
```

This example is very similar to the crc32 one, because both crc32 and sha1 implement the hash.Hash interface. The main difference is that whereas crc32 computes a 32 bit hash, sha1 computes a 160 bit hash. There is no native type to represent a 160 bit number, so we use a slice of 20 bytes instead.

Servers

Writing network servers in Go is very easy. We will first take a look at how to create a TCP server:

```
package main

import (
  "encoding/gob"
  "fmt"
  "net"
)
```

```
func server() {
  // listen on a port
  ln, err := net.Listen("tcp", ":9999")if
  err != nil {
    fmt.Println(err)
    return
  }
  for {
    // accept a connectionc,
    err := ln.Accept() if err
    != nil { fmt.Println(err)
    continue
    }
    // handle the connection
    go handleServerConnection(c)
  }
}
func handleServerConnection(c net.Conn) {
  // receive the message
  var msg string
  err := gob.NewDecoder(c).Decode(&msg)if
  err != nil {
    fmt.Println(err)
  } else { fmt.Println("Received",
    msg)
  }

  c.Close()
}

func client() {
  // connect to the server
  c, err := net.Dial("tcp", "127.0.0.1:9999")if
  err != nil {
    fmt.Println(err)
    return
  }

  // send the message msg
  := "Hello World"
  fmt.Println("Sending", msg)
  err = gob.NewEncoder(c).Encode(msg)if
  err != nil {
    fmt.Println(err)
  }

  c.Close()
}
func main() { go server() go client()
var input string fmt.Scanln(&input)
```

}
HTTP

HTTP servers are even easier to setup and use:
package main

import ("net/http" ; "io")

func hello(res http.ResponseWriter, req *http.Request) { res.Header().Set(
"Content-Type", "text/html",
)
io.WriteString(res,
`<DOCTYPE html>
<html>
<head>
<title>Hello World</title>
</head>
<body>
Hello World!
</body>
</html>`,
)
}
func main() { http.HandleFunc("/hello", hello) http.ListenAndServe(":9000", nil)
}

HandleFunc handles a URL route (/hello) by calling the given function. We can also handle
static files by using FileServer:
http.Handle("/assets/", http.StripPrefix("/assets/",
http.FileServer(http.Dir("assets")),
),
)

The (remote procedure call) and
RPC net/rpc

packages provide an easy way to expose
methods so they can be invoked over a network net/rpc/jsonrpc t in the program running them)

package main

import (
"fmt"
"net" "net/rpc"
)

type Server struct {}
func (this *Server) Negate(i int64, reply *int64) error {
*reply = -i
return nil
}

```
func server() {
  rpc.Register(new(Server))
  ln, err := net.Listen("tcp", ":9999")
  if err != nil {
    fmt.Println(err)
    return
  }
  for {
    c, err := ln.Accept()
    if err != nil {
    continue
    }
    go rpc.ServeConn(c)
  }
}
func client() {
  c, err := rpc.Dial("tcp", "127.0.0.1:9999")
  if err != nil {
    fmt.Println(err)
    return
  }
  var result int64
  err = c.Call("Server.Negate", int64(999), &result)
  if err != nil {
    fmt.Println(err)
  } else {
    fmt.Println("Server.Negate(999) =", result)
  }
}
func main() {
  go server()
  go client()

  var input string
  fmt.Scanln(&input)
  }
```

This program is similar to the TCP example, except now we created an object to hold all the methods we want to expose and we call the Negate method from the client. See the documentation in net/rpc for more details.

Parsing Command Line Arguments

When we invoke a command on the terminal it's possible to pass that command arguments. We've seen this with the go command:

```
go run myfile.go
```

run and myfile.go are arguments. We can also pass flags to a command:

```
go run -v myfile.go
```

The flag package allows us to parse arguments and flags sent to our program. Here's an example program that generates a number between 0 and 6. We can change the max value by sending a flag (-max=100) to the program:

```
package main
import ("fmt";"flag";"math/rand") func main() {
// Define flags
maxp := flag.Int("max", 6, "the max value")
// Parse flag.Parse()
// Generate a number between 0 and max fmt.Println(rand.Intn(*maxp))
}
```

Any additional non-flag arguments can be retrieved with flag.Args() which returns a []string.

Synchronization Primitives

The preferred way to handle concurrency and synchronization in Go is through go routines and channels as discussed in chapter 10. However Go does provide more traditional multithreading routines in the `sync` and `sync/atomic` packages.

Mutexes: A mutex (mutal exclusive lock) locks a section of code to a single thread at a time and is used toprotect shared resources from non-atomic operations. Here is an example of a mutex:

```
package main

import ( "fmt"
"sync"
"time"
)

func main() {
m := new(sync.Mutex)

for i := 0; i < 10; i++ { go func(i int) { m.Lock() fmt.Println(i, "start")
time.Sleep(time.Second) fmt.Println(i, "end") m.Unlock()
}(i)
}

var input string fmt.Scanln(&input)
```

}When the mutex (m) is locked any other attempt to lock it will block until it is unlocked. Great care should be taken when using mutexes or the synchronization primitives provided in the package.

Traditional multithreaded programming is difficult; it's easy to make mistakes and those mistakes are hard to find, since they may depend on a very specific, relatively rare, and difficult to reproduce set of circumstances. One of Go's biggest strengths is that the concurrency features it provides are much easier to understand and use properly than threads and locks.