Chapter 5

Objective

- Interfaces in Go
- Declaring and implementing an interface
- Interface internal representation
- Empty Interface
- Type Switch
- Deferred Functions Calls
- File Processing In Go Language
- Making Of A File
- Streaming IO in golang
- Write Bytes to a File
- Command Line Arguments
- Enumerating arguments
- Write Slice To A File
- Read File into a Slice
- Use OS Package for Standard I/O
- Use io Package To Read a File
- Use io Package to Write A String To A File
- The ioUtil package
- Read File Line by Line Using bufio

This page is left blank intentionally

Unit 1

Interfaces in Go

- An interface in golang is a way to achieve **Polymorphism**
- The general definition of an interface in the Object oriented world is "interface defines the behavior of an object".
- Behavior only specifies what the object is supposed to do.
- In golang interface is a collection of method signatures that an object can implement.
- Therefore interface defines the behavior of the object.
- A variable of interface type can hold any value that implements these methods.
- When a type provides definition for all the methods in the interface, it is said to implement the interface.
- Interface specifies what methods a **type** should have and the type decides how to implement these methods.

For example:

Shopping can be an interface with method signatures BuyGroceries() and BuyCloths().

```
type Shopping interface {
    BuyGroceries()
    BuyCloths()
}
```

- Any type which provides definition for BuyGroceries() and BuyCloths() is said to implement the Shopping interface.

```
type Groceries struct {
   Fruit string
   Vegetables string
   Meat string
}

type Cloths struct {
   WinterCloths string
   SummerCloths string
}

func (g Groceries) BuyGroceries() {

}

func (c Cloths) BuyCloths()

}
```

Declaring and implementing an interface

- Before declaring an interface, first need to create a method that is attached to **struct type** or **non struct**
- See chapter #4 unit #2, how to attach method to **struct type** or **non struct**

```
package main // Example 5-1
import (
   "fmt"
   "math"
)

type Shape interface {  // New Code
   Area() float64       // New Code
}

type Rectangle struct {
   length float64
   width  float64
}

type Circle struct {
   radius float64
}

func (r Rectangle) Area() float64 { // Receiver is r
   return r.length * r.width
}

func (c Circle) Area() float64 { // Receiver is c
   return math.Pi * c.radius * c.radius
}

```

```
func main() {
   r := Rectangle{
      length: 10,
      width:  5,
   }

   // fmt.Printf("Area of rectangle %d\n", r.Area())

   c := Circle{
      radius: 12,
   }

   // fmt.Printf("Area of circle %f\n", c.Area())

   shapeArea := Shape.Area(r)          // New Code
   fmt.Printf("Rectangle Area %f\n", shapeArea)  // New Code

   shapeArea = Shape.Area(c)   // New Code
   fmt.Printf("Circle Area %f\n", shapeArea)  // New Code
}
```

Output:

```
Rectangle Area 50.000000
Circle Area 452.389342
```

```
package main // Example 5-2
import (
   "fmt"
   "unicode" // Unicode - universel character encoding standard.
)

// interface type VowelsFinder with one method
type VowelsFinder interface {
   FindVowels() []rune
}

type MyString string // non struct type

// Add method FindVowels()[]rune to the receiver type string
func (ms MyString) FindVowels() []rune {

  // MyString is said to implement the interface VowelsFinder
   var vowels []rune
   for _, rune := range ms {
      switch unicode.ToLower(rune) {
      case 'a', 'e', 'i', 'o', 'u':
         vowels = append(vowels, rune)
      }
   }
   return vowels
}

func main() {
   name := MyString("Game Of Thrones")
   var v VowelsFinder

  v = name // possible since MyString implements VowelsFinder
  fmt.Printf("Vowels in %s are %c\n", name, v.FindVowels())
  fmt.Printf("v Type: %T and name Type: %T\n", v, name)
}
```

```
Output:
        Vowels in Game Of Thrones are [a e O o e]
        v Type: main.MyString and name Type: main.MyString
```

Unit 2

Interface internal representation

- An interface can be thought of as being represented internally by a type and value.
- The type is the underlying concrete type of the interface and value holds the value of the concrete type.

```go
package main // Example 5-3
import "fmt"

// Test interface has one method Tester() and MyFloat type implements that interface.
type Test interface {
   Tester()
}

type MyFloat float64   // non struct type

func (m MyFloat) Tester() {
   fmt.Println("Inside Tester Body: ", m)
}

func describe(t Test) {
   fmt.Printf("Interface type %T value %v\n", t, t)
}

func main() {
   var t Test // Test is an interface type
   f := MyFloat(89.7)

  // assign the variable f of type MyFloat to t which is of type Test.
   t = f
  // concrete type of t is MyFloat and the value of t is 89.7
   describe(t)
   t.Tester()
}
```

Output:

```
Interface type main.MyFloat value 89.7
Inside Tester Body:  89.7
```

Empty Interface

- An interface which has zero methods is called empty interface.
- It is represented as interface{}.
- The empty interface has zero methods and therefore all types implement the empty interface.

```go
package main   // Example 5-4
import "fmt"

// function takes an empty interface as argument, pass to any type
func describe(i interface{}) {
    fmt.Printf("Type = %T, value = %v\n", i, i)
}

func main() {
    s := "Hello World"
    describe(s)  // pass string
    i := 55
    describe(i)  // pass integer
    strt := struct {
        name string
    }{
        name: "John",
    }

    describe(strt)
}

```

Output:

```
Type = string, value = Hello World
Type = int, value = 55
Type = struct { name string }, value = {John}
```

Unit 3

Type Switch

- Use type switch to discover the type of an interface value.
- A switch can be used to discover the dynamic type of an interface variable.
- A type switch uses the syntax of a type assertion with the keyword type inside the parentheses.
- Type assertion is an operation applied to the value of the interface.
- A type switch compares types instead of values.

```
package main // Example 5-5
import "fmt"

type Employee struct {
   name string
   age  int
}

func main() {
   emp := Employee{"Suleman",45}

   checkType := func(i interface{}) {
      switch t := i.(type) {
      case bool:
         fmt.Printf("boolean Type value =  %t\n", t)
      case int:
         fmt.Printf("integer Type value = %d\n", t)
      case *int:
         fmt.Printf("pointer to integer Type value = %d\n", *t)
      case string:
         fmt.Printf("String Type value = %s\n", t)
      case Employee:
         fmt.Println("Employee struct")
         fmt.Println("Name:", t.name, "Age:", t.age)
      default:
         fmt.Printf("Don't know type %T\n", t)
      }
   }
   checkType(true)
   checkType(20)
   checkType("Hello")
   checkType(emp)
}
```

```
Output:
      boolean Type value =  true
      integer Type value = 20
      String Type value = Hello
      Employee struct
      Name: Suleman Age: 45
```

Unit 4

Deferred Functions Calls

- A **defer keyword** schedules a function call to be run after the function completes.
- A **defer keyword** ensure that resources are released in all cases, regardless of complexity of the control flow.
- The correct way to use defer keyword is immediately after the resource has been acquired.
- The **defer keyword** is often used with the paired operations for the resource such as:
    - ➢ open and close
    - ➢ connect and disconnect
    - ➢ lock and unlock

```
package main   // Example 5-6
import "fmt"

func main() {
    fmt.Println("Hello")
    for i := 1; i <= 3; i++ {
        defer fmt.Println(i)
    }
    fmt.Println("World")
}
```

Output

```
Hello
World
3
2
1
```

Unit 5

File Processing In Go Language

- Variables and arrays are used for temporary storage of data in internal memory
- Files are used for permanent storage of large amounts of data on a secondary storage devices (usually some type of disk or tape)
- Files are organized for sequential access or random access (also called direct access)
- A file can contain formatted data (as would be written to the monitor), or unformatted "raw" data (as it is stored in memory)

Making Of A File

- A group of related bytes is called a field
- A group of related fields is called a record; in golang, we can represent these as a **struct**
- Usually one field in each record is chosen as a key field that uniquely identifies the record
- A group of related records is a file
- A group of related files is a database

Streaming IO in Go

- The Go **io** package provides interfaces **io.Reader** and **io.Writer**, for data input and output operations

- Golang comes with many APIs that support streaming IO from resources like in-memory structures, files, network connections, and many other

- Type **os.File** represents a file on the local system.

- It implements both **io.Reader** and **io.Writer** and, therefore, can be used in any streaming IO contexts.

- Use **io.Reader** (and **io.Writer**) whenever you’re dealing with streams of data.

Write Bytes to a File

Using os package bytes can be written to a file

Go provides other file I/O packages such as: **io, ioutil, and bufio**

Every package that is imported will increase the size of executable.

```go
package main // Example 5-7
import (
   "log"
   "os"
)

func main() {
   // Open a new file for writing only
   file, err := os.OpenFile("test.txt",
      os.O_WRONLY|os.O_TRUNC|os.O_CREATE,
      0666,
   )
   if err != nil {
      log.Fatal(err)
   }

   defer file.Close()

   // Write bytes to file
   byteSlice := []byte("Bytes!\n")

   bytesWritten, err := file.Write(byteSlice)
   if err != nil {
      log.Fatal(err)
   }

   log.Printf("Wrote %d bytes.\n", bytesWritten)
}

```

Output:

```
2020/10/15 22:53:06 Wrote 7 bytes.
```

Unit 6

Command Line Arguments

- CLI, or "command line interface," is a program that users interact with on the command line.
- CLI programs expect some input in the form of CLI arguments.
- Program arguments are handled in golang as a slice of strings:

```
var Args []string
```

- Retrieving the name of the currently running program

```
package main // Example 5-8
import (
   "fmt"
   "os"
)

func main() {
    // Program Name is always the first (implicit) argument
    cmd := os.Args[0]

    fmt.Printf("Program Name: %s\n", cmd)
}

```

- Determine the Number of Arguments Passed
- Remember that the first argument is always the program name
- Use the slice by using os.Args[1:], does not include program name
- It will give new subslice starting with index 1 (not 0) to the end of the slice.

```
package main // Example 5-9
import (
   "fmt"
   "os"
)

func main() {
   argCount := len(os.Args[1:]) // without program name
   fmt.Printf("Total Arguments : %d\n", argCount)
}

```

Enumerating arguments

```go
package main // Example 5-10
 import (
    "fmt"
    "os"
 )

 func main() {
    for i, arg := range os.Args[1:] {
       fmt.Printf("Argument %d is %s\n", i+1, arg)
    }
 }

/*
Running the program with ./example5-10 -local u=admin --help
Argument 1 is -local
Argument 2 is u=admin
Argument 3 is --help
*/
```

Unit 7

Write Slice To A File

- Write successive string slices directly to a file:

```go
package main // Example 5-11
import (
   "fmt"
   "os"
)

func main() {
   proverbs := []string{
     "The golang language development started in 2007\n",
     "The main authors of golang are:\n",
     "Robert Griesemer, Rob Pike and Ken Thompson\n",
     "The golang will change the world, just like C did.\n",
   }
   file, err := os.Create("./proverbs.txt")
   if err != nil {
      fmt.Println(err)
      os.Exit(1)
   }
   defer file.Close()

   for _, p := range proverbs {
      n, err := file.Write([]byte(p))
      if err != nil {
         fmt.Println(err)
         os.Exit(1)
      }
      if n != len(p) {
         fmt.Println("failed to write data")
         os.Exit(1)
      }
   }
   fmt.Println("file write done")
}

```

Read File into a Slice

- Type io.File can be used as a reader to stream the content of a file from the local file system.

```go
package main // Example 5-12
import (
   "fmt"
   "os"
   "io"
)

func main() {
   file, err := os.Open("./proverbs.txt")
   if err != nil {
      fmt.Println(err)
      os.Exit(1)
   }
   defer file.Close()

   p := make([]byte, 4)
   for {
      n, err := file.Read(p)
      if err == io.EOF {
         break
      }
      fmt.Print(string(p[:n]))
   }
}

```

Use OS Package for Standard I/O

- The os package exposes three variables, **os.Stdout**, **os.Stdin**, and **os.Stderr**, that are of type ***os.File**

- Print string directly to the standard output using os.Stdout

```
package main // Example 5-13
import (
   "fmt"
   "os"
)

func main() {
   proverbs := []string{
     "The golang language development started in 2007\n",
     "The main authors of golang are:\n",
     "Robert Griesemer, Rob Pike and Ken Thompson\n",
     "The golang will change the world, just like C did\n",
   }

   for _, p := range proverbs {
      n, err := os.Stdout.Write([]byte(p))
      if err != nil {
         fmt.Println(err)
         os.Exit(1)
      }
      if n != len(p) {
         fmt.Println("failed to write data")
         os.Exit(1)
      }
   }
}
```

Use io Package To Read a File

- The io.Copy lets you read ALL bytes from an io.Reader, and write it to an io.Writer
- Reads from a file and prints to standard output using the io.Copy() function as shown below:

```
package main // Example 5-14
import (
   "fmt"
   "os"
   "io"
)

func main() {
   file, err := os.Open("./proverbs.txt")
   if err != nil {
      fmt.Println(err)
      os.Exit(1)
   }
   defer file.Close()

   if _, err := io.Copy(os.Stdout, file); err != nil {
      fmt.Println(err)
      os.Exit(1)
   }
}
```

Use io Package to Write A String To A File

```go
package main // Example 5-15
import (
   "fmt"
   "os"
   "io"
)

func main() {
   file, err := os.Create("./magic_msg.txt")
   if err != nil {
      fmt.Println(err)
      os.Exit(1)
   }
   defer file.Close()
   if _, err := io.WriteString(file, "Golang is fun!"); err != nil {
      fmt.Println(err)
      os.Exit(1)
   }
}

```

The ioUtil package

- Package ioutil, a sub-package of io, offers several convenience functions for **I/O**.
- Use function ReadFile to load the content of a file into a []byte.

```
package main // Example 5-16
import (
   "fmt"
   "os"
   "io/ioutil"
)

func main() {
   bytes, err := ioutil.ReadFile("./planets.txt")
   if err != nil {
      fmt.Println(err)
      os.Exit(1)
   }
   fmt.Printf("%s", bytes)
}
```

Read File Line by Line Using bufio

- The **bufio** is a package used for buffered IO
- Buffering IO is a technique used to temporarily accumulate the results for an IO operation **before** transmitting it forward.
- This technique can increase the speed of a program by reducing the number of system calls, which are typically slow operations.
- Using **NewScanner** function from **bufio** will split the text into token

```go
package main // Example 5-17
import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    // Open the file
    fileHandle, _ := os.Open("emp.dat")
    defer fileHandle.Close()

    // Create a new Scanner for the file
    fileScanner := bufio.NewScanner(fileHandle)

    // Loop over all lines in the file and print them.
    for fileScanner.Scan() {
        fmt.Println(fileScanner.Text())
    }
}

```

```
Cameron Wu;29;50589
Clifton Stillman;65;99900
John Kaufman;53;69597
Kurt Lamm;39;90000
Larry Godwin;45;59500
Patrick Stroud;48;140565
Paul Goldsmith;60;200000
Susan Carltom;42;85000
Ursula Spencer;27;36450
William Reynolds;37;77550
```