Day-2

A tour of Go

- Every Go program is made up of packages. Programs start running in package `main`
- Import statements are like this

  ▼

  ```
  import "fmt"

  import {
              "fmt"
              "math/rand"
  }
  ```

- When importing a package, only exported names can be used. Exported names start with a capital letter.

  ▼

  ```
  fmt.Println(math.pi) // this will cause error
  fmt.Println(math.Pi) // this works perfectly fine
  ```

- Declaration statements are read left to right. variable name comes before datatype.

  ▼

  ```
  x int // int
  p *int // pointer to int
  a [3]int // array[3] of int
  ```

  ```
  func add(x int, y int) int {
          return x+y
  }
  //Go allows defining types of similar variables togethe
  func add(x, y, z int) int {
          return x + y + z
  }
  ```

- Function can return any number of results.

  ▼

  ```
  func swap(x, y string) (string, string) {
          return y, x
  }

  func main() {
          a, b := swap("hello", "world!")
          fmt.Println(a,b)
  }// output : world! hello
  ```

- Go's return values may be named. They are treated as variables defined at the top of the function. A `return` statement without arguments returns the named return values. This is known as ***naked return***.

  ▼

```
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}

func main() {
    fmt.Println(split(17))
}
// output: 7 10
```

- The `var` statement declares a list of variables. A `var` statement can be at package or function level.

  ▼

  ```
  var c, python, java bool

  func main() {
      var i int
      fmt.Println(i, c, python, java)
  }
  ```

- A var declaration can include initializers, one per variable. If an initializer is present, the type can be omitted.

  ▼

  ```
  var i, j int = 1, 2

  var c, python, java = true, false, "Yes!"

  func main() {
          fmt.Println(i, j, c, python, java)
  }
  // output: 1 2 true false Yes!
  ```

- Inside a function, the := short assignment statement can be used in place of a `var` declaration with implicit type. Outside a function, every statement begins with a keyword(`var` , `func` and so on) and so := construct is not available.

▼

```
func main() {
	var i, j int = 1, 2
	k := 3
	c, python, java := true, false, "No!"

	fmt.Println(i, j, k, c, python, java)
}
```

- Go's basic types are given below. The `int`, `uint` and `uintptr` types are usually 32-bits wide on 32-bit systems and 64 bits wide on 64-bit systems.

▼

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32
     // represents a Unicode code point

float32 float64

complex64 complex128
```

```
var (
    ToBe   bool       = false
    MaxInt uint64     = 1<<64 - 1
    z      complex128 = cmplx.Sqrt(-5 + 12i)
)
```

```go
func main() {
    fmt.Printf("Type: %T Value: %v\n", ToBe, ToBe)
    fmt.Printf("Type: %T Value: %v\n", MaxInt, MaxInt)
    fmt.Printf("Type: %T Value: %v\n", z, z)
}

// Output:
// Type: bool Value: false
// Type: uint64 Value: 18446744073709551615
// Type: complex128 Value: (2+3i)
```

- The expression `T(v)` converts the value `v` to the type `T` . In Go assignment between items or different type requires an explicit conversion.

  ▼

```go
func main() {
    var x, y int = 3, 4
    var f float64 = math.Sqrt(float64(x*x + y*y))
    var z uint = uint(f)
    fmt.Println(x, y, z)
}
// Output: 3 4 5
```

- Constants are declared with `const` keyword. They can be of any type int, bool, string etc. But const can not be declared using := syntax.

  ▼

```go
const Pi = 3.14

func main() {
    const World = "世界" // world in chinese
    fmt.Println("Hello", World)
    fmt.Println("Happy", Pi, "Day")

    const Truth = true
    fmt.Println("Go rules?", Truth)
}
Output:
// Hello 世界
// Happy 3.14 Day
// Go rules? true
```

- Numeric constants are high-precision values.

  ▼

  ```
  const (
      // Create a huge number by shifting a 1 bit left 10
      // In other words, the binary number that is 1 foll
      Big = 1 << 100 // 2^100
      // Shift it right again 99 places, so we end up wit
      Small = Big >> 99 // 2
  )

  func needInt(x int) int { return x*10 + 1 }
  func needFloat(x float64) float64 {
      return x * 0.1
  }

  func main() {
      fmt.Println(needInt(Small))
      fmt.Println(needFloat(Small))
      fmt.Println(needFloat(Big))
  }
  Output:
  // 21
  // 0.2
  // 1.2676506002282295e+29
  ```

- There is only one looping construct, the `for` loop. The basic loop has 3 components similar to C language without parentheses surrounding three components and the braces are always required.

  ▼

```
func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum)
}
```

- `While` equivalent in Go is

  ▼

  ```
  sum := 1
  for sum < 1000 {
          sum+=sum
  }
  fmt.Println(sum)
  ```

- Similar to the for loop syntax, `if` statements need not be surrounded by `()` but the braces `{}` are required.

  ▼

  ```
  func sqrt(x float64) string {
      if x < 0 {
          return sqrt(-x) + "i"
      }
      return fmt.Sprint(math.Sqrt(x))
  }

  func main() {
      fmt.Println(sqrt(2), sqrt(-4))
  }
  ```

- Interesting pattern regarding execution order of `fmt.print` statements.

▼

```
func pow(x, n, lim float64) float64 {
    fmt.Println("pow called with",x,n,lim)
    if v := math.Pow(x, n); v < lim {
        return v
    } else {
        fmt.Printf("%g >= %g\n", v, lim)
    }
    // can't use v here, though
    return lim
}

func main() {
    fmt.Println(
        pow(3, 2, 10),
        pow(3, 3, 20),
    )
}
```

Guess the output:

```
Ouput:
pow called with 3 2 10
pow called with 3 3 20
27 >= 20
9 20
```