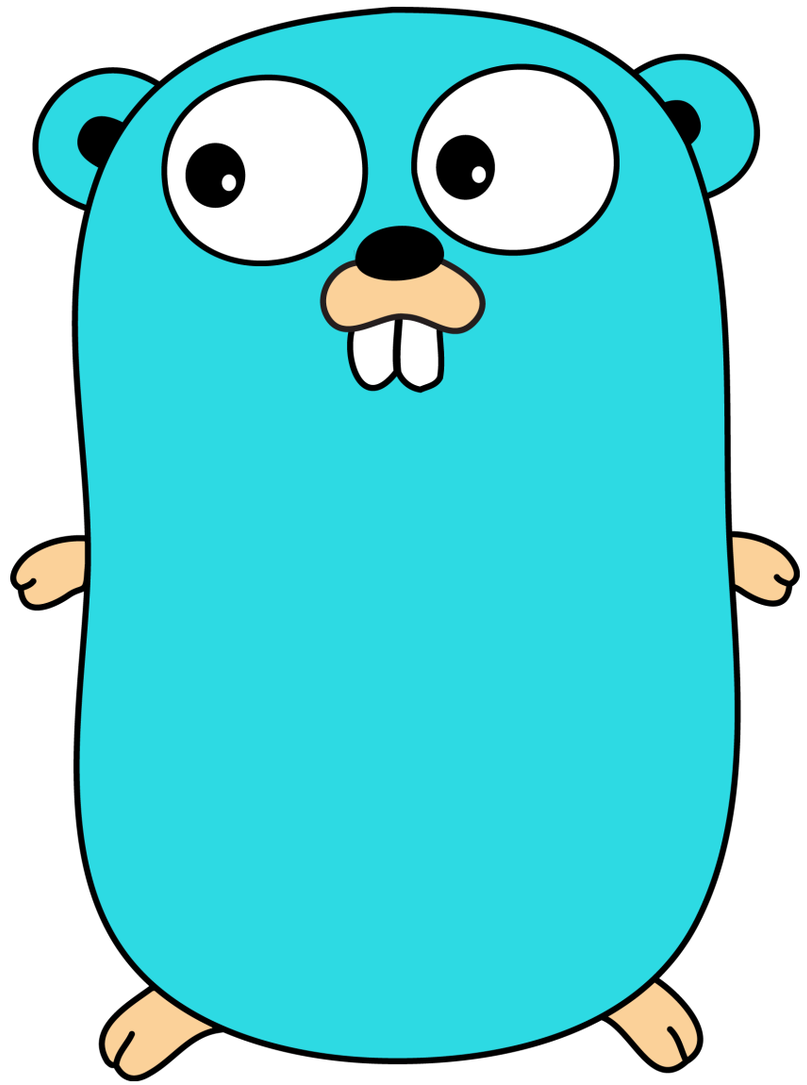

A Tour of Go

Jin Hyun Kim

Fall 2019

This Slides from

- https://tour.golang.org/

- http://golang.site/

In this class

- We will cover most of features of Go language so that you can create a small piece of program.

Hello World

- Install Go lang
- Install GoLanD from https://www.jetbrains.com/idea/
- Create “helloworld” project
- Run

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello, World")
7 }
```

Run Go

- Compile and run
  - $ go build xxx.go
  - $ xxx
- Just run
  - $ go run xxx.go

Beginning

Package

```go
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println("My favorite number is",
rand.Intn(10))
}

```

- The package *main* is is a directive for compilers

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Printf("Now you have %g problems.\n",
math.Sqrt(7))
}

```

- *import* read in a package

Function

```go
package main

import "fmt"

func add(x int, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(42, 13))
}
```

- Notice how to declare a variable and a function
  - Function type - Return value type
  - Function parameters

In this example, we shortened

```
x int, y int
```

to

```
x, y int
```

Function

```
1  package main
2
3  import "fmt"
4
5  func swap(x, y string) (string, string) {
6      return y, x
7  }
8
9  func main() {
10     a, b := swap("hello", "world")
11     fmt.Println(a, b)
12 }
13
```

- Multiple results can be returned

Function

```
1  package main
2
3  import "fmt"
4
5  func split(sum int) (x, y int) {
6      x = sum * 4 / 9
7      y = sum - x
8      return
9  }
10
11 func main() {
12     fmt.Println(split(17))
13 }
14
15
```

- Named results
    - Go's return values may be named. If so, 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 a "naked" return.

Variables

```go
package main

import "fmt"

var c, python, java bool

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

- Use keyword *var* to declare variable types

```go
package main

import "fmt"

var i, j int = 1, 2

func main() {
    var c, python, java = true, false, "no!"
    fmt.Println(i, j, c, python, java)
}

```

- Initialize variables

Variables

```go
package main

import "fmt"

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

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

- Inside a function, the := short assignment statement can be used in place of a var declaration with implicit type.
- Outside a function,
  - := construct is not available.

Basic Types

```
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
```

Basic Types

```go
package main

import (
    "fmt"
    "math/cmplx"
)

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

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)
}
```

Basic Types

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var i int
7      var f float64
8      var b bool
9      var s string
10     fmt.Printf("%v %v %v %q\n", i, f, b, s)
11 }
12
```

- The zero value is:
  - **0** for numeric types,
  - **false** for the boolean type, and
  - **""** (the empty string) for strings.

Type Conversions

```go
package main

import (
    "fmt"
    "math"
)

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)
}
```

- The expression T(v) converts the value v to the type T.

Some numeric conversions:

```go
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

Or, put more simply:

```go
i := 42
f := float64(i)
u := uint(f)
```

Type Inference

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     v := 42 // change me!
7     fmt.Printf("v is of type %T\n", v)
8 }
```

- When declaring a variable without specifying an explicit type (either by using the := syntax or var = expression syntax),

- the variable's type is inferred from the value on the right hand side.

Type Inference

```
var i int
j := i // j is an int
```

- When the right hand side of the declaration is typed, the new variable is of that same type

```
i := 42           // int
f := 3.142        // float64
g := 0.867 + 0.5i // complex128
```

- When the right hand side contains an untyped numeric constant,
  - the new variable may be an int, float64,
    or complex128 depending on the precision of the constant:

Constants

```go
package main

import "fmt"

const Pi = 3.14

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

    const Truth = true
    fmt.Println("Go rules?", Truth)
}
```

- A constant variable can be one of character, string, and boolean

- Constants cannot be declared using the := syntax.

Constants

```
package main

import "fmt"

func main() {
    var x, y, z int = 1, 2, 3
    c, python, java := true, false, "no!"

    fmt.Println(x, y, z, c, python, java)
}
```

- Initialize constant variables

Numeric Constants

```
package main

import "fmt"

const (
    // Create a huge number by shifting a 1 bit left 100 places.
    // In other words, the binary number that is 1 followed by 100 zeroes.
    Big = 1 << 100
    // Shift it right again 99 places, so we end up with 1<<1, or 2.
    Small = Big >> 99
)

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))
}

```

- Numeric constants are high-precision valu es.
- An untyped constant takes the type needed by its context.

Now, Walk on Go (Control)

For

```go
package main

import "fmt"

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

```go
package main

import "fmt"

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

- Iteration
- Go Lang have no more iteration than *for*
    - The basic for loop has three components separated by semicolons:
        - the *init* statement: executed before the first iteration
        - the *condition* expression: evaluated before every iteration
        - the *post* statement: executed at the end of every iteration

Like while

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      sum := 1
7      for sum < 1000 {
8          sum += sum
9      }
10     fmt.Println(sum)
11 }
```

- For is Go's "while"

Eternal Loop

```go
package main

func main() {
    for {
    }
}
```

- The same as

```python

while(True):
    print("Hello")
```

- in Python

if

```go
package main

import (
    "fmt"
    "math"
)

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))
}
```

- Conditional statements
- The expression need not be surrounded by parentheses ( ) but the braces { } are required.

if

```go
package main

import (
    "fmt"
    "math"
)

func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}

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

- Like for, the if statement can start with a short statement to execute before the condition.
- Variables declared by the statement, eg., v, are only in scope until the end of the if.
- Question: Explain how the result comes out

if-else

```
package main

import (
    "fmt"
    "math"
)

func pow(x, n, lim float64) float64 {
    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),
    )
}

```

- Variables declared inside an if short statement are also available inside any of the else blocks.

switch

```go
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan9, windows...
        fmt.Printf("%s.\n", os)
    }
}
```

- *Switch* statement in Go only runs the selected case, not all the cases that follow

- Switch cases evaluate cases from top to bottom, stopping when a case succeeds.

switch

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
    }
}
```

- Switch without a condition is the same as switch true.
- This construct can be a clean way to write long if-then-else chains.

switch with no condition

```
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
    }
}
```

- This construct can be a clean way to write long if-then-else chains.

defer

- A *defer* **statement** defers the execution of a function until the surrounding function returns.

```
package main

import "fmt"

func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
}
```

```
package main

import "os"

func main() {
    f, err := os.Open("1.txt")
    if err != nil {
        panic(err)
    }

    // main 마지막에 파일 close 실행
    defer f.Close()

    // 파일 읽기
    bytes := make([]byte, 1024)
    f.Read(bytes)
    println(len(bytes))
}
```

defer

```go
package main

import "fmt"

func main() {
    fmt.Println("counting")

    for i := 0; i < 10; i++ {
        defer fmt.Println(i)
    }

    fmt.Println("done")
}
```

- Deferred function calls are pushed onto a stack. When a function returns, its deferred calls are executed in last-in-first-out order.

panic

```go
package main

import "os"

func main() {
    openFile("Invalid.txt")
    println("Done") //이 문장은 실행 안됨
}

func openFile(fn string) {
    f, err := os.Open(fn)
    if err != nil {
        panic(err)
    }
    // 파일 close 실행됨
    defer f.Close()
}
```

- Go 내장함수인 panic()함수는 현재 함수를 즉시 멈추고 현재 함수에 defer 함수들을 모두 실행한 후 즉시 리턴

Recover

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    openFile("1.txt")
    println("Done") // 이 문장 실행됨
}

func openFile(fn string) {
    // defere 함수. panic 호출시 실행됨
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("OPEN ERROR", r)
        }
    }()

    f, err := os.Open(fn)
    if err != nil {
        panic(err)
    }

    // 파일 close 실행됨
    defer f.Close()
}
```

- Go 내장함수인 recover()함수는 panic 함수에 의한 패닉상태를 다시 정상상태로 되돌리는 함수

Variable Type

```go
package main

import (
    "fmt"
    "math/cmplx"
)

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

func main() {
    const f = "%T(%v)\n"
    fmt.Printf(f, ToBe, ToBe)
    fmt.Printf(f, MaxInt, MaxInt)
    fmt.Printf(f, z, z)
}
```

const f = "%T(%v)\n"

```
bool


string


int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr


byte // uint8의 다른 이름(alias)


rune // int32의 다른 이름(alias)
     // 유니코드 코드 포인트 값을 표현합니다.


float32 float64


complex64 complex128
```

Variable Types

http://pyrasis.com/book/GoForTheReallyImpatient/Unit41

```
package main

import "fmt"

func main() {
     var num1 int = 10
     var num2 float32 = 3.2
     var num3 complex64 = 2.5 + 8.1i
     var s string = "Hello, world!"
     var b bool = true
     var a []int = []int{1, 2, 3}
     var m map[string]int = map[string]int{"Hello": 1}
     var p *int = new(int)
     type Data struct{ a, b int }
     var data Data = Data{1, 2}
     var i interface{} = 1

     fmt.Println(num1) // 10: 정수 출력
     fmt.Println(num2) // 3.2: 실수 출력
     fmt.Println(num3) // (2.5+8.1i): 복소수 출력
     fmt.Println(s)    // Hello, world!: 문자열 출력
     fmt.Println(b)    // true: 불 출력
     fmt.Println(a)    // [1 2 3]: 슬라이스 출력
     fmt.Println(m)    // map[Hello:1]: 맵 출력
     fmt.Println(p)    // 0xc0820062d0: 포인터(메모리 주소) 출력
     fmt.Println(data) // {1 2}: 구조체 출력
     fmt.Println(i)    // 1: 인터페이스 출력

     fmt.Println(num1, num2, num3, s, b) // 10 3.2 (2.5+8.1i) Hello, world! true
     fmt.Println(p, a, m)                // 0xc0820062d0 [1 2 3] map[Hello:1]
     fmt.Println(data, i)                // {1 2} 1
}
```

Pointer

```go
package main

import "fmt"

func main() {
    i, j := 42, 2701

    p := &i          // point to i
    fmt.Println(*p) // read i through the pointer
    *p = 21          // set i through the pointer
    fmt.Println(i)  // see the new value of i

    p = &j          // point to j
    *p = *p / 37   // divide j through the pointer
    fmt.Println(j) // see the new value of j
}
```

- Go has pointers. A pointer holds the memory address of a value.

- The type *T is a pointer to a T value. Its zero value is nil.

Pointer

- Go has pointers. A pointer holds the memory address of a value.
- The type *T is a pointer to a T value. Its zero value is nil.

```
var p *int
```

- The & operator generates a pointer to its operand.

```
fmt.Println(*p) // read i through the pointer p
*p = 21         // set i through the pointer p
```

- The * operator denotes the pointer's underlying value.

```
i := 42
p = &i
```

Struct

```go
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    fmt.Println(Vertex{1, 2})
}

```

- A struct is a collection of fields.

Struct

```go
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    v.X = 4
    fmt.Println(v.X)
}
```

- Struct fields are accessed using a dot.

Struct

```go
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    p := &v
    p.X = 1e9
    fmt.Println(v)
}
```

- Struct fields can be accessed through a struct pointer

- To access the field X of a struct when we have the struct pointer p we could write **(*p).X**

- **p.X** is allowed without the explicit dereference.

Struct Literals

```
1  package main
2
3  import "fmt"
4
5  type Vertex struct {
6      X, Y int
7  }
8
9  var (
10     v1 = Vertex{1, 2}  // has type Vertex
11     v2 = Vertex{X: 1}  // Y:0 is implicit
12     v3 = Vertex{}      // X:0 and Y:0
13     p  = &Vertex{1, 2} // has type *Vertex
14 )
15
16 func main() {
17     fmt.Println(v1, p, v2, v3)
18 }
```

- A struct literal denotes a newly allocated struct value by listing the values of its fields.

- You can list just a subset of fields by using the Name: syntax. (And the order of named fields is irrelevant.)

Arrays

- The type **[n]T** is an array of n values of type T.

```
var a [10]int
```

- An array's length is part of its type, so arrays cannot be resized.

Slices

```go
package main

import "fmt"

func main() {
    primes := [6]int{2, 3, 5, 7, 11, 13}

    var s []int = primes[1:4]
    fmt.Println(s)
}

```

- The type **[]T** is a slice with elements of type T.

- A slice is formed by specifying two indices, a low and high bound, separated by a colon:

```
a[low : high]
```

```
a[1:4]
```

Slices

```go
package main

import "fmt"

func main() {
    names := [4]string{
        "John",
        "Paul",
        "George",
        "Ringo",
    }
    fmt.Println(names)

    a := names[0:2]
    b := names[1:3]
    fmt.Println(a, b)

    b[0] = "XXX"
    fmt.Println(a, b)
    fmt.Println(names)
}
```

- A slice does not store any data, it just describes a section of an underlying array.

- Changing the elements of a slice modifies the corresponding elements of its underlying array.

Slice literals

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      q := []int{2, 3, 5, 7, 11, 13}
7      fmt.Println(q)
8
9      r := []bool{true, false, true, true, false, true}
10     fmt.Println(r)
11
12     s := []struct {
13         i int
14         b bool
15     }{
16         {2, true},
17         {3, false},
18         {5, true},
19         {7, true},
20         {11, false},
21         {13, true},
22     }
23     fmt.Println(s)
24 }
25
```

- A slice literal is like an array literal without the length.

```
[3]bool{true, true, false}
```

```
[]bool{true, true, false}
```

Slice Defaults

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      s := []int{2, 3, 5, 7, 11, 13}
7
8      s = s[1:4]
9      fmt.Println(s)
10
11     s = s[:2]
12     fmt.Println(s)
13
14     s = s[1:]
15     fmt.Println(s)
16 }
```

- When slicing, you may omit the high or low bounds to use their defaults instead.

```
a[0:10]
a[:10]
a[0:]
a[:]
```