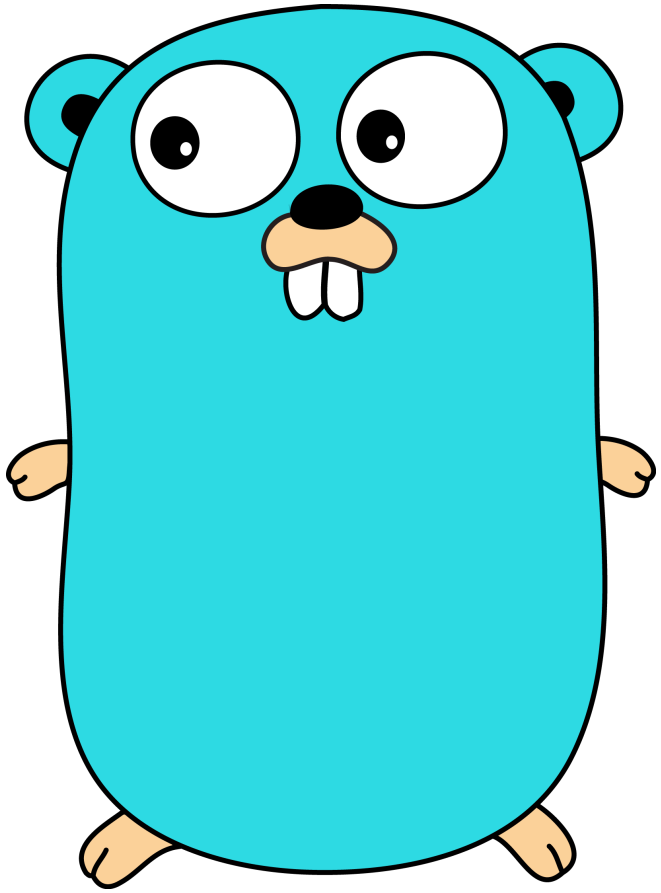

A Tour of Go

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 pieces of software

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"
)

func main() {
    fmt.Println("Happy", math.Pi, "Day")
}
```

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

```go
package main

import "fmt"
import "math"

func main() {
    fmt.Printf("Now you have %g problems.",
        math.Nextafter(2, 3))
}
```

- *import* read in a package

Function

```
package main

import "fmt"

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

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

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

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

Function

```go
package main

import "fmt"

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

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

- Multiple results can be returned

Function

```go
package main

import "fmt"

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

func main() {
    fmt.Println(split(17))
}
```

- Named results

Variables

```go
package main

import "fmt"

var x, y, z int
var c, python, java bool

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

- Use keyword *var* to declare variable types

```go
package main

import "fmt"

var x, y, z int = 1, 2, 3
var c, python, java = true, false, "no!"

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

- Initialize variables

Constants

```go
package main

import "fmt"

const Pi = 3.14

func main() {
    const World = "안녕"
    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

```
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 (
    Big = 1 << 100
    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 values.
- 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)
}
```

- Iteration
- Go Lang have no more iteration than *for*

Like while

```
package main

import "fmt"

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

- For is Go's "while"

Eternal Loop

```go
package main

import "fmt"

func main() {
    for {
        fmt.Println("Hello")
    }
}
```

- The same as

```python

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

- in Python

if - else

```
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 - else

```
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),
    )
}
```

```
27 >= 20
9 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

switch

```
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
  - In effect,
    the *break* statement that is needed at the end of each case in those languages is provided automatically in Go

switch

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("When's Saturday?")
    today := time.Now().Weekday()
    switch time.Saturday {
    case today + 0:
        fmt.Println("Today.")
    case today + 1:
        fmt.Println("Tomorrow.")
    case today + 2:
        fmt.Println("In two days.")
    default:
        fmt.Println("Too far away.")
    }
}
```

```
switch i {
case 0:
case f():
}
```

does not call `f` if `i==0`.)

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

```
package main

import "fmt"

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

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

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

```
1  package main
2
3  import "os"
4
5  func main() {
6      f, err := os.Open("1.txt")
7      if err != nil {
8          panic(err)
9      }
10
11     // main 마지막에 파일 close 실행
12     defer f.Close()
13
14     // 파일 읽기
15     bytes := make([]byte, 1024)
16     f.Read(bytes)
17     println(len(bytes))
18 }
```

defer

```
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
```