Workshop: Introduction To Go

24 July 2015

Kenji Pa
Cat Developer, Oursky

Why Go?

- Compiled language
- *Feels* like scripting language

```
a, b := aComplex.Components()    // YEAH!
```

- Explicit, tracable code.
- Simple language. Don't try to do everything. You know exactly what can and *cannot* be done.
- Built-in concurrency
- Great toolings, like `go get`, `go install`, `got fmt`, `golint`, etc...

Okay, I'm in. How to get started?

- Install Go binary: golang.org/doc/install (http://golang.org/doc/install)
- Set up GOPATH: golang.org/doc/code.html (http://golang.org/doc/code.html)

tl;dr

```
$ brew install go
$ mkdir ~/go
$ EXPORT GOPATH=~/go
$ EXPORT PATH='$PATH;~/go/bin'
```

Okay we can write some programs

YOU KNOW THE DEAL

OUR FIRST GO PROGRAM

Hello World

```
package main // this is an executable package

import "fmt" // we need to use this package! (aka. library)

func main() { // every executable should have one and only one main
    fmt.Println("Hello World!") // boring stuff
}
```

Run

HELLO WORLD IS BORING

COME ON, JAMES

```
package main

import "fmt"

func main() {
    var james string // note the variation declaration
    // TODO(limouren): receive user input here
    james = "James"
    fmt.Println("Comes on, " + james + "!")

    // declare with initializer
    // type definition is not need
    excuse := "money"
    fmt.Println("I'm not talking about", excuse)
}
```

Zero value

- When you declare a variable like this:

```
var aVariable Type
```

`aVariable` will be assigned a "zero value".

Zero value of builtin type:

- Numbers: 0
- string: ""
- bool: `false`
- struct: zero value of its members
- Pointer: nil

Functions

```
func add(a, b int) int { return a + b }

// multiple return values
func divmod(a, b int) (int, int) { return a / b, a % b }

// function being first-class citizen
var minus = func(a, b int) int { return a - b }

func main() {
    var a, b, c, d int
    a = 1
    b = 2

    c = add(a, b)
    fmt.Printf("a + b = %d\n", c)

    // variable unpacking
    c, d = divmod(a, b)
    fmt.Printf("a divmod b = (%d, %d)\n", c, d)

    fmt.Printf("a - b = %d\n", minus(a, b))
}
```

Run

Variable Declaration

- You know those basics

```
var (
    boolValue bool = true
    stringValue string = "string"
    intValue int = 1 // int8, int16, int32, int64, uint, uint8 and so on...
    floatValue float32 = 1.0 // float64
    complexValue complex32 = builtin.complex(0, 1)
)
```

Flow Control

Flow Control: if

```
if cond {
  // cond is true
} else {
  // cond is false
}
```

Flow Control: Loop

Of course!

```
package main

import "fmt"

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

Run

Flow Control: While

Hmm...?

```
package main

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

func main() {
    age := 0
    deathAt := rand.Intn(100)
    for age < deathAt {
        fmt.Print(".")
        time.Sleep(10 * time.Millisecond)
        age++
    }
    fmt.Println("")

    fmt.Printf("Died at %d.\n", age)
}
```

Run

Flow Control: For'ever

Frankly, you only need `for`

```
for {
  // runs forever...
}
```

`for` rocks!

Flow Control: Switch

```
var name string
fmt.Scanf("%s", &name)

// implicit break in each cases
switch name {
case "Edwin":
  fmt.Println("handsome!")
case "Carmen":
  fmt.Println("pretty!")
case "Chima", "Faseng":
  fmt.Println("cute!")
default:
  fmt.Println("boring!")
}
```

Data Types

19.1. Struct

- Delcare

```
type Location struct {
```

```
    lat, lon float64
}
```

- Instantiate

```
loc := Location{lat: 1, lon: 1}
```

More Data Types

Array

We have

```
var integers [8]int
```

Declare with initializer:

```
integers = [8]int{0, 1, 2, 3, 4, 5, 6, 7}
```

But fixed-length array isn't very helpful. Of course... Go got you covered.

Slice

Dynamic array is called "slice". To Declare:

```
integers []int
```

Initialize, like array:

```
integers := []int{0, 1, 2, 3, 4, 5, 6, 7}
```

Accessing member:

```
integers[5] == 5
```

Slice (cont.)

"Slice" a slice:

```
// integers[1:4] == []int{1, 2, 3}
```

Getting its length:

```
len(integers) == 8
```

Slice that cannot grow is useless:

```
integers = append(integers, 9, 10)
```

C's guys will cry out if they cannot specify the initial length and capacity of slice:

```
len := 9000
cap := 9001
integers = make([]int, len, cap)
// it's Over 9000!
```

Map

So the type looks like:

```
map[KeyType]ValueType
```

To declare it:

```
var m map[string]int
```

To initialize it:

```
m1 := make(map[string]int)
m2 := map[string]int{}
// the two statements are functionally equivalent
```

Initialize with some item:

```
wordCount := map[string]int{
  "word": 1,
  "map": 7,
}
```

Map (cont.)

Set item:

```
wordCount["vocabulary"] = 6
```

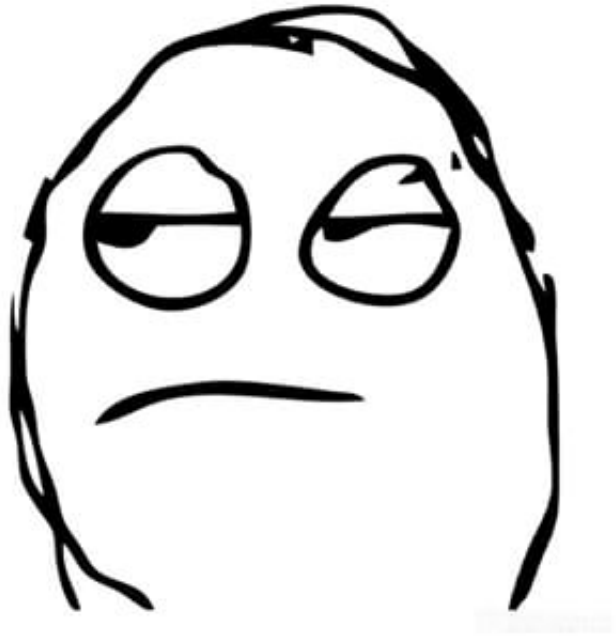
Delete item:

```
delete(wordCount, "vocabulary")
```

Access items:

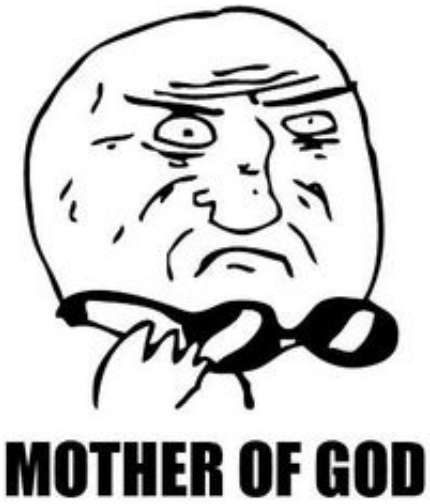
```
i := wordCount["word"]
// i == 7

j := wordCount["not-exist"]
// access a key that doesn't exist return the zero value of ValueType
// in this case j == 0

k, ok := wordCount["not-exist"]
// use the ok pattern to check key existence
// ok == false
```

Man it's boring



Give me classes and inheritance and polymorphism!

Go doesn't have Class



Method

There are no classes in Go, but you can write methods on `struct`:

```
package main

import "fmt"

type Triangle struct {
    Base, Height float64
}

func (t Triangle) Area() float64 {
    return t.Base * t.Height / 2.0
}

func main() {
    t := Triangle{Base: 3, Height: 4}
    fmt.Println(t.Area())
}
```

Run

Method (cont.)

```
func (t Type) Method()
```

`t` is called *receiver*, which refers to the struct this method is acting. Instead of Type, method can also be declared on a pointer *Type:

```
func (tp *Type) Method()
```

The difference between `pointer receiver` and non-pointer receiver is that pointer receiver can modify the receiver: `tp` is pointing to the actual struct being called on.

Method (cont.)

```
package main

import "fmt"

type Triangle struct {
    Base, Height float64
}

func (t Triangle) Area() float64 {
    return t.Base * t.Height / 2.0
}

func (t *Triangle) Scale(by float64) {
    t.Base *= by
    t.Height *= by
}

func main() {
    t := Triangle{Base: 3, Height: 4}
    t.Scale(2)
    fmt.Println(t.Area())
}
```

Run

Q: Isn't it Class?

No.

You cannot *subclass* a struct. Struct is struct, a collection of data and nothing more.

Reuse code by composition. :P

Q: So Go can't write my favourite `Shape` examples? Go is useless for me!

Yes and No. Stay tuned.

Interface

- Interface is a type that declares a set of methods.
- It can be used to hold any value that implemented all methods defined in an interface.
- Much like Java's `interface`
- Declare a interface:

```
type Interface interface {
  Method(t1 Type1, t2 Type2) Type3
}
```

Interface (Cont.)

```
package main

import "fmt"

type Animal interface {
    Speak()
}

type Dog struct{}

func (dog Dog) Speak() {
    fmt.Println("Woooooooooooooooooooof~")
}

type Cat struct{}

func (cat Cat) Speak() {
    fmt.Println("Meow~")
}

func main() {
    var animal Animal

    animal = Dog{}
    animal.Speak()
```

```
    animal = Cat{}
    animal.Speak()
}
```

Run

Interface (Cont.)

Some observations:

- `Cat` and `Dog` do not need to explicitly state that they implement `Animal`. As long as they has a method called Speak(), they satified the requirement being a `Animal`.
- Go's interface-only approach emphasizes on the *behaviour* of an object. It's about *How this object behaves* instead of *What is this object*
- `Cat` is more adorable than `Dog`.

Interface: Exercise

Implement the classic *Shape* example in Go:

```
package main

import "fmt"

// Shape defines the methods that a shape should have
type Shape interface {
    // Name returns the name of the Shape
    Name() string

    // Area returns the area of the Shape
    Area() float64
}

func main() {
    shapes := []Shape{
        Rectangle{Width: 10, Height: 2},
        Triangle{Base: 3, Height: 4},
        Circle{Radius: 3},
    }

    for _, shape := range shapes {
        fmt.Printf("%s.Area() = %v\n", shape.Name(), shape.Area())
```

```
    }
}

type Rectangle struct {
    Width, Height float64
}

// implement Rectangle methods here

type Triangle struct {
    Base, Height float64
}

// implement Triangle methods here

type Circle struct {
    Radius float64
}

// implement Circle methods here
```

Run

Interface (Cont.)

Some builtin interfaces you might want to look into

- `fmt.Stringer`: makes your own type printable
- `io.Reader` and `io.Writer`: streaming io
- `http.Handler`: self-descriptive. Used to write http handler.

Concurrency

Go's concurrency is built on two components:

- goroutine
- channel

Goroutine

Goroutine is a function that can be run concurrently

Thank you

Kenji Pa
Cat Developer, Oursky
limouren@gmail.com (mailto:limouren@gmail.com)
http://www.facebook.com/limouren (http://www.facebook.com/limouren)
@limouren (http://twitter.com/limouren)