Learn the Go Programming Language

For experienced developers or
those of an adventurous nature

gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

Level 03

Intro to Types
Composite Types

`v0.5 draft`

Types

- Go is statically typed
- No type casting (everything is type conversion)
- All types have a "zero" value:

```
string   ""
numbers  0
pointers nil
bool     false
```

Type elision

- Only within functions, shorthand declaration:

```
v := getSomething()
// same as
var v TYPE
v = getSomething()
```

- This one simple feature is a big part of how go makes static typing less painful.

Zero Value

- All memory in go is initialized.

```
// initialized to “thing”
s2 := “thing”

// initialized to “”
var s2 string

// initialized to 0.0
var f float32

// initialized to false
var isOk bool
```

Type Conversions

- This is not type casting
- This only works for a types that the compiler knows can be directly converted

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0+0.0i of type complex128
string('x')              // "x" of type string
string(0x266c)           // "♬" of type string
MyString("foo"+"bar") // "foobar" of type MyString
```

Type Assertions

```
v, ok := unknownThing.(<TYPE>)

    if ok {
    // v has a type of <TYPE>,
    // v is assigned unknownThing
} else {
    // v has a type of <TYPE>
    // v is nil or zero value
}
```

Type Switches

```
switch safelyTypedVariable := unknownType.(type) {
    case nil:
        printString("unknownType is nil")
    case int:
        printInt(safelyTypedVariable)      // safelyTypedVariable is an int
    case float64:
        printFloat64(safelyTypedVariable)  // safelyTypedVariable is a float64
    case func(int) float64:
        printFunction(safelyTypedVariable) // safelyTypedVariable is a function
    case CustomString, string:
        // safelyTypedVariable is still of some unknown type
        printString("type is CustomString or string")
    case []string:
        // safelyTypedVariable is an array of strings
    case []interface{}:
        // safelyTypedVariable is an array of interface{} (basically anything)
    default:
        printString("don't know the type")
}
```

Composite Types

- structs
- array
- slice
- map

Arrays

- Fixed, Declared length and type

```
// array of 3 integers, all initialized to 0
var array_of_ints [3]int
```

- use the built-in len() to get size

```
len(array_of_ints) // is 3
```

- Arrays are values in go

- We rarely use arrays in go because we have slices

Slices

- We use these all the time
- A slice is a reference to a section of an array
- Looks like an array without the length declaration

```
// slice of integers
var slice_of_ints []int
```

- Also uses len()
- If you want to create a slice of an existing array you can do so:

```
slice_of_ints := array_of_ints[1:2]

len(slice_of_ints) // is 1 (the second element from array_of_ints)
```

Slicing

- When slicing, first index defaults to 0:
  `aoi[:n] means the same as aoi[0:n]`

- Second index defaults to len(array/slice):
  `aoi[n:] means the same as aoi[n:len(aoi)]`

- Thus to create a slice from an array:
  `aoi[:] means the same as aoi[0:len(aoi)]`

Making Slices

- Slice Literal (create a slice and it's underlying array)

```
var slice = []int{1,2,3,4,5}
```

- Slice Allocations (allocate a slice and underlying array with the built-in function `make`)

```
// returns []int with 10 zeroed ints
var intSlice = make([]int, 10)

// returns []int with capacity for 20 ints total
var intSlice = make([]int, 0, 20)

// returns []int with 10 zeroed ints and capacity for 20 ints total
var intSlice = make([]int, 10, 20)

// returns *[]int
var intSlicePtr = new([]int)
```

Growing a Slice

- Use the built-in append function:

```
append(s []Type, x ...Type) []Type
```

```
slice := []int{1, 2, 3}
slice = append(slice, 4)
slice = append(slice, 5, 6, 7)
```

- When possible, append will grow the slice in place
- Append is very cheap when you do the proper capacity planning ahead of time
- Use the built-in cap() function to see the capacity of the underlying array

Slice Misc

- Slices are references and are very cheap to make and pass around
- care must be taken as underlying storage (array) can be modified out from underneath you
- Strings can also be sliced with similar efficiency

Maps

- maps aka kv, dictionaries, associative arrays, hash tables, etc.
- in go map types are references
- len() returns the number of keys
- This is a map name m where the key is a string and the value is a float

```
var m map[string]float64
```

Making Maps

- Literal: list of colon-separated key:value pairs

```
m = map[string]float64{"one":1, “pi":3.1415}
empty_map = map[string]float64{}
```

- make()

```
m := make(map[string]float64) // make empty map
```

- declare and assign

```
var m1 map[string]float64
m1[“three"] = 3 // CRASH!
```

Using Maps

```
m := map[string]int{“one”:1, “two”:2}

// indexing a map
x = m[“one”]          // x is 1
x = m[“no such key”] // x is 0 (zero value)

// check existence
x, exists := m[key]
if exists == true {
    // key exists, x is assigned the value for key
} else {
    // key DNE, x is the zero value
}

// removing key/values
delete(m, “two”)
// if map or key is nil, delete is a no-op
```

Iterating Collections

- Special range keyword
- arrays and slices

```
for index, value := range array_or_slice {
    //...
}
```

- maps

```
for key, value := range some_map {
    //...
}
```

- Can use _ (blank identifier) if you don't care about index or key or value

```
for _, value := range array_or_slice {
    //...
}
```

Pointers

- go has pointers, but no pointer math
- go has nil
- use new() to make a pointer to anything

```
var pp *Point = new(Point)
```

- or use the address operator &

```
var pp *Point = &Point{}
```

Make() Vs New()

- slices, maps & channels use make()

- new() always makes pointers

Reference vs. Value

- Everything in go is pass by value
- Some values are reference values
  - eg. slices, maps and of course, pointers
- sometimes passing small structs as values is cheaper than pointers
  - Use the profiler! Use the benchmark tool!

Structs

- Fairly obvious at first glance:

```
type Point struct {
    x int
    y int
}
var p Point
```

- Structs are values

- Init'd to all zero values

- always use the dot notation to access fields

```
p.x = 1
p.y = 2
```

Making Structs

- Typically you either declare for a value or use new() for a pointer (reference)

```
var p Point       // a new struct
pp := new(Point) // a new pointer to a struct
var pp2 *Point   // a new pointer, to nil
```

- Struct Literals

```
p := Point{1,2}
p2 := Point{y:2}
p3 := Point{
   y:2,
   x:1,
}

pp := &Point{1,2}
pp := &Point{}  // same as new(Point)
```

Visibility

- Capital initial letter field & method means exported visibility
- Lowercase initial letter means unexported

```
package “zoology”
type Animal struct {
    Name string
    genus string
}
```

Go 101 ☑

- At this point, you have a basic understand of the go syntax and it's low level building blocks
    - basic syntax, numbers, strings, expressions, flow control, slices, maps, structs
- The Real Power™ of go comes after this:
    - Composition via methods, anonymous fields and interfaces
    - Concurrency via goroutines and channels
    - The great advances made in the compilers and associated tooling
    - The breath and depth of the standard library

Thank You, Credits & License

http://gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

- Much of the content is inspired by (and in some cases, outright taken from) a CCA3.0 Licensed (http://creativecommons.org/licenses/by/3.0/us/), 3 day Go Course by Rob Pike that predates Go 1.0 and is considered **out of date:**
    - http://go.googlecode.com/hg-history/release-branch.r60/doc/GoCourseDay1.pdf
    - http://go.googlecode.com/hg-history/release-branch.r60/doc/GoCourseDay2.pdf
    - http://go.googlecode.com/hg-history/release-branch.r60/doc/GoCourseDay3.pdf

- I owe many many, thanks to the many authors of Go and to Rob Pike in particular.