Learn the Go Programming Language

For experienced developers or
those of an adventurous nature

gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

2

LEVEL 01

The Basic Basics:
Syntax Intro, Numbers, Strings

v0.5 draft

Selling the Dream

- Go will minimize **time to production**
- Easiest language to refactor existing code
- Overall a tremendous productivity multiplier
- Syntax, tooling good for teams, without penalizing individuals
- See Level 0 for more info

Go is not for everyone, but those who take the time deftly wield the nuances of the language will find their reward is astonishingly good productivity.

Syntax at a Glance

- C-like, but a bit more modern
- No semicolons
- Compiler enforced brace style (like K&R or 1TBS)
- Consistent formatting (gofmt)

http://en.wikipedia.org/wiki/Indent_style#K.26R_style

Syntax Misc.

- Comments:

```
/* This is a comment; no nesting */
// So is this.
```

- Identifiers are letters and numbers (plus '_') with "letter" and "number" defined by Unicode.

LITERALS

- Number literals just are (no size or type suffixes)

```
23
0x0FF
1.234e7
```

- Strings: double quoted

```
"Hello, world\n"
"\xFF"    // 1 byte
"\u00FF"  // 1 Unicode char, 2 bytes of UTF-8
```

- Raw & multi-line strings: backtick'd

```
`\n\.abc\t\` == "\\n\\.abc\\t\\"
`multi
line` == “multi\nline"
```

backticks are like “”” in python

KEYWORD NAME TYPE

- Declarations are of the form: <KEYWORD> <NAME> <TYPE>

```
var i int
var pi, pj *int  // note difference from C
var numbers []int
const PI = 22./7.
type S struct { a, B int }
type Thinger interface { … }
func check() error { … }
```

- Capitalization denotes exported/unexported
  - Struct S above is exposed to packages that import this code
  - Struct S itself has one private (a) and one public (B) field

KNT order Contrast w/ C which is somewhat inconsistent

exported/unexported is analagous to public/private.

The bit on capitalization feels weird at first, but there's a certain elegance and grace once you get used to it.

Note that unexported is just semi-private. If you are feeling dangerous, you can access private fields via reflection and other means.

GROUPING KEYWORDS

```
var (
    i int
    j = 356.245
    k int = 0
    l, m uint64 = 1, 2
    nanoseconds int64 = 1e9
    inter, floater, stringer = 1, 2.0, "hi"
)

// also works for const, type (not func)
```

IOTA

```
// iota is an enumeration-like type

type Month int
const (
      January Month = iota
      February
      March
      // ...
)

// the above is equivalent to:
const January Month = 0
const February Month = 1
const March Month = 2
```

http://golang.org/ref/spec#Iota

IOTA

```
type Shape int
const (
    // iota starts at 0
    Triangle Shape = iota + 3 // Triangle  == 3
    Rectangle                 // Rectangle == 4
    Pentagon                  // Pentagon  == 5
)

const (
    a = 1 << iota  // a == 1 (iota has been reset)
    b = 1 << iota  // b == 2
    c = 1 << iota  // c == 4
)

const (
    no_pi = iota * 3.14159265359  // no_pi  == 0
    pi                            // pi     == 3.14159265359
    two_pi                        // two_pi == 6.28318530718
)
```

http://golang.org/ref/spec#Iota

http://play.golang.org/p/ehPcFdslWI

Quickly on the Type System

- Go is statically typed
- No type casting (everything is type conversion)
- Type Elision

http://golang.org/ref/spec#Conversions

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.

You will use and love this.

HELLO.GO

namespaced

```
package main

import "fmt"

func main() {
    fmt.Print("Hello, 世界\n")
}
```

no header files

Everything is UTF-8

NUMBERS

```
int,  int8,  int16,  in32,    in64
uint, uint8, uint16, uint32,  uint64
      byte
                     float32, float64
                              complex64, complex128
```

the size of int and uint are architecture-dependent. Typically they represent a "native" integer on a specific architecture/platform.

More on Numbers

- byte is uint8 under the hood
- int is not the same type as int32, even on 32-bit systems
- In order to prevent subtle errors, you must always convert numeric types manually
- numeric type conversion will overflow, truncate and round:
  - http://golang.org/ref/spec#Conversions
- constants are mathematically "exact"

Historically, google has many, many runtime errors related to numeric type casting

int may be the same size as int32 on 32-bit systems, but that is just an implementation detail

Numeric Constants

- A decimal or exponent denotes floating point.

```
1.234e5   // floating-point
1e2       // floating-point
3.2i      // imaginary floating-point
100       // integer
077       // octal integer
0xFEEDBEEEEEEEEEEEEEEEEEEF  // hexadecimal integer
```

- Can mix and match numerical literals:

```
2*3.14  // floating point: 6.28
3./2    // floating point: 1.5
3/2     // integer: 1
3+2i    // complex: 3.0+2.0i
```

MATHEMATICALLY "EXACT"

- No L or U or UL suffixes.
- By exact, we mean internal implementation is excessive.
- Current spec guarantees:
  - integer: at least 256 bits
  - floating-point: mantissa of at least 256 bits and a signed exponent of at least 32 bits
  - compiler will error on int or fp overflow, round for fp precision

http://golang.org/ref/spec#Constants

I believe the internal representation of float constants is around 1024 bits.

pre-1.0, the go team used to call these ideal constants, but that wasn't exactly true. They were more ideal than most other compilers at the time, but

Bool

- `bool`
- `false` and `true` are bool values
- In order to prevent an entire class of errors, you can never use a pointer or integer when a bool type is expected
  - if statements, etc.

STRING

- `string`
- Length-delimited, not null-terminated
- Under the hood, arrays of bytes
- Immutable
  - you can reassign a string variable, but "hello" is always "hello"
- Standard library has all the goodies:
  - `strings, path, url, regex, etc.`
- distinct from the type `[]byte`

just like 3 is always 3, "hello" is always "hello"

If you come from cocoa, you have NSString and NSData

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.
- These slides are Copyright 2013-2014 Matthew Nunogawa
- All content is licensed under the Creative Commons Attribution 4.0 License (http://creativecommons.org/licenses/by/4.0/)
  - attribution: Matt Nunogawa, Copyright 2013-2014 Matthew Nunogawa, http://gotutorial.net
- All code is licensed under a BSD License (http://opensource.org/licenses/BSD-2-Clause)

These are the slides that I used to learn go back in 2011.
"out of date": The actually syntax has not significantly changed. Some of the terminology is no longer in use, typically because after contact with the community, misunderstandings have occurred.

In the creation of these slides, I have, to the utmost of my ability, attempted to make sure that these are correct and updated. Any errors are likely my fault. I make no guarantee that these slides are correct or will remain correct under the inevitable progression of time.