Learn the Go Programming Language

For experienced developers or
those of an adventurous nature

gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

LEVEL 02

Go with the Flow:
Expressions, Statements & Flow Control

v0.5 draft

EXPRESSIONS

```
// binary operators
// Mostly C-like
// higher line means higher order of operation

* / % << >> & &^
+ - | ^
== != < <= > >=
&&
||

// Unary operators
& ! * + - ^ <-
```

&^ means bit clear

EXPRESSIONS

- Surprises for the C programmer:
  - fewer precedence levels (should be easy)
  - ^ instead of ~ (it's binary "exclusive or" made unary)
  - ++ and -- are not expression operators
    - (x++ is a statement, not an expression;*p++ is (*p)++ not *(p++))
  - &^ is new; handy in constant expressions
  - << >> etc. require an unsigned shift count

This slide copied verbatim from Go Day 1 course

EXPRESSIONS

- Non-surprises:
  - assignment ops work as expected: += <<= &^= etc.
  - expressions generally look the same (indexing, function call, etc.)

This slide copied verbatim from Go Day 1 course

Control Flow

- if
  - like C, but no parens, and also has an optional "init" statement
- for
  - like C but quite bit more flexible
  - no `while` or `do…while` in go, can use `for` instead
- switch
  - waaaay more flexible than the C version
- select
  - look for our slides on channels for discussion on this one

IF

- Stop me if you've seen this before:

```
if x < 5 { less() }
if x < 5 { less() } else if x == 5 { equal() }
```

- 2 statement variant (init statement)

```
if x := getX(); x < 5 { … }
if result, err := maybe(); err != nil { … }
```

For

- Stop me if you've seen this before:

```
for i := 0; i < 10; i++ { ... }
```

- missing statements resolve to true:

```
for ; ; { /* infinte loop */ }
for { /* infinte loop */ }
```

- Can do more than one thing at a time:

```
for i,j := 0,N; i < j; i,j = i+1,j-1 {...}
```

There is no comma operator as in C

SWITCH

- case statements can be anything (unlike C)
- fallthrough is optional and explicit
- cases can be comma-separated

In C, your case statements are basically limited to integers and constants

SWITCH: CONSTANTS

```
switch get() {
    case 0:     // no-op
    case 4,5,6: msg = “safe”
    case 10:    fallthrough
    case 1,11:  msg = “danger”
    default:    msg = “invalid”
}
```

In C, your case statements are basically limited to integers and constants

SWITCH: EXPRESSIONS

```
a, b := getPair()
switch {
    case a < b: …
    case a == b: …
    case a > b: …
}

// nearly same as

switch a, b := getPair(); {
    case a < b: …
    case a == b: …
    case a > b: …
}
```

don't forget the semi-colon in the second example!

"nearly same as": scope is different for a,b

http://play.golang.org/p/KTdw9HuxPK

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

Type Switches

- If you get an unknown type (discussed in level 3) there is a specific use case of the switch statement that allows you convert to a specific safely typed variable

```
switch safelyTypeVar := unknownType.(type) {
    case nil:
        printString("unknownType is nil")
    case int:
        printInt(safelyTypeVar)      // safelyTypeVar is an int
    case float64:
        printFloat64(safelyTypeVar)  // safelyTypeVar is a float64
    case func(int) float64:
        printFunction(safelyTypeVar) // safelyTypeVar is a function
    case []string:
                                     // safelyTypeVar is a slice of strings
   default:
        printString("don't know the type")
}
```

more on type switches and conversion in level 3

Break, Continue, Labels

- Basically same as C
- Use Labels to be more explicit with breaks:

```
JustTenThenGiveUpLoop:
    for i := 0; i < 10; i++ {
        num :=  f()
        switch {
        case num < 1<<30:
            break JustTenThenGiveUpLoop
        }
        fmt.Println(i, num)
    }
```

http://play.golang.org/p/4r1MYH6SPR

vs

http://play.golang.org/p/3nXcdeVyDD

GOTO

- Yes, go has `goto`

FUNCTIONS

- Basic form:

```
func run() { … }
func square(f float64) float64 { … }
func squareRoot(f float64) (float64, error) { … }
func getMsg() (msg string, err error) { … }
```

Blank Identifier

```
func squareRoot(f float64) (float64, error) { … }

func main() {
    // use the blank identifier (_) for don't cares
    _, err := squareRoot(1)
}
```

In this case, we just want to check the error. We aren't particularly interested in the value.

Named Return Values

Can help w/ understandability

```
func DidSomething() (success bool) { … }
func GetMsg() (msg string, err error) {
   msg = get()
   if msg == “” {
      return “invalid”, fmt.Errorf(“get() failure”)
   }
   return
}
```

initialized w/ zero value

initialized w/

You don't have to return the variables.

a "naked" return means return use the named values

DEFER

- defer will execute a function when the enclosing function returns

```
func DoSomething() string {
    r := resource.Open()
    defer resource.Close()

    if num := r.GetInt(); num < 3 {
       return “one, two”
    } else if num == 3 {
       return “tree!”
    }
    return “wtf”
}
```

NESTED DEFER

- multiple defers will execute in LIFO order

```
func DoSomething() string {
    for i := 0; i < 5; i++ {
        defer fmt.Printf("%d ", i)
    }
}
```

Args Now, Function Later

- Arguments execute immediately, the deferred function executes upon return

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}
func un(s string) {fmt.Println("leaving:", s)}
func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}
func b() {
    defer un(trace("b")) fmt.Println("in b") a()
}
func main() { b() }
```

This is advanced go!

http://play.golang.org/p/-ZinZqGWNF

Function Literals

- All Function Literals are closures
- Fairly straightforward:

```
// assign to g
g := func(i int) { fmt.Printf("%d",i) }
g(i)

// or just execute a function straightaway:
func(i int) {
    fmt.Printf(“%d",i)
}()
```

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 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.