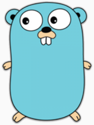
Programming in Go
Lesson 2: Packages & Functions

Matt Holiday

23 April 2019

Cardinal Peak



Lesson #2

What we'll cover today:

- Homework #1
- Slice gotchas
- IDEs
- Packages
- Functions, parms, & returns
- Scope of variables
- Gotchas with `:=`
- Closures
- Defer

Homework #1: First program

```go
package main

import (
	"fmt"
	"io/ioutil"
	"os"
	"strings"
)

func main() {
	var n int

	// don't use range here, you don't want the first arg!

	for i := 1; i < len(os.Args); i++ {
		fn := os.Args[i]
		text, err := ioutil.ReadFile(fn)
```

Homework #1: First program

```
        // handle the case of a bad file name

        if err != nil {
            fmt.Fprintf(os.Stderr, "can't read %s: %s\n",
                        fn, err)
            continue
        }

        // magic happens here
        // we must convert the []byte from ReadFile

        words := strings.Fields(string(text))
        n += len(words)
    }

    fmt.Println(n, "total words")
}
```

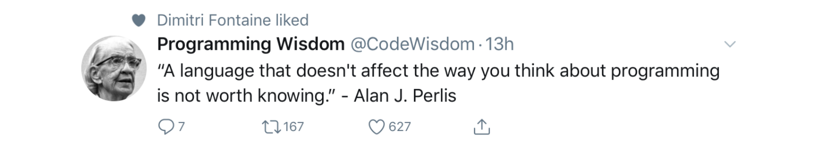
Homework #1: Second program

```go
func main() {
    m := make(map[string]int)  // can't just use var

    for i := 1; i < len(os.Args); i++ {
        fn := os.Args[i]
        text, err := ioutil.ReadFile(fn)

        if err != nil { /* error handling here */ }

        words := strings.Fields(string(text))

        for _, w := range words {  // ignore keys
            m[w] += 1  // m[w] returns 0 on first access
        }
    }

    fmt.Println(len(m), "unique words")
}
```

Homework #1: Results

```
## file rich2.txt is taken from Shakespeare, Richard II act 2 scene 1

$ go run counter1.go rich2.txt
2558 total words

$ wc rich2.txt
     372    2558   14102 rich2.txt

$ go run counter2.go rich2.txt
1197 unique words

$ awk '{for (i=1; i<=NF; i++) {print $i}}' rich2.txt|sort|uniq|wc
    1197    1197    7993

$ awk '{for (i=1; i<=NF; i++) {print $i}}' rich2.txt|sort|uniq|head -5
&
'Gainst
'Tis
'gainst
'mongst
```

“Understanding nil”



Slice Gotchas

Slice follow-up

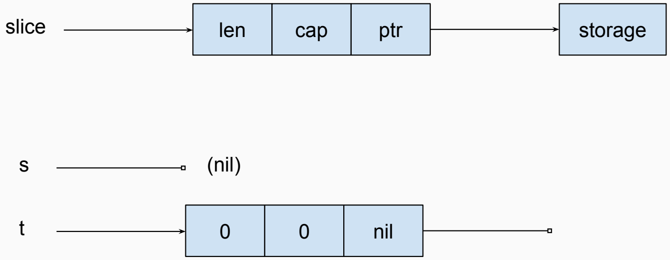
play.golang.org

The Go Playground

```go
package main

import (
	"fmt"
)

func main() {
	var s []int
	var t = []int{}

	fmt.Printf("%d, %d, %#v %t\n", len(s), cap(s), s, s == nil)
	fmt.Printf("%d, %d, %#v %t\n", len(t), cap(t), t, t == nil)
}
```

```
0, 0, []int(nil) true
0, 0, []int{} false

Program exited.
```

Slice follow-up



Ugly #1: Slice length vs capacity

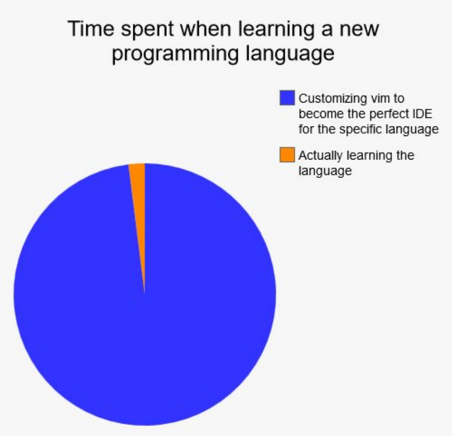
```go
// let's make an array of 3 items

a := [3]int{1, 2, 3}

b := a[0:1]          // b is a slice of a's first item

fmt.Println(b)       // prints [1]

c := b[0:2]          // WTF? but the array has 3 entries

fmt.Println(c)       // prints [1 2]

fmt.Println(len(b))  // prints 1
fmt.Println(cap(b))  // prints 3

b := a[0:1:1]        // this is what you probably meant
```

Ugly #2: Slice mutating underlying array

```go
a := [3]int{1, 2, 3}
b := a[0:1]; c := b[0:2]

b = append(b, 4)        // grows b, mutates a
fmt.Println("a=",a)   // a= [1 4 3]
fmt.Println("b=",b)   // b= [1 4]

c = append(c, 5)        // grows c, mutates a
fmt.Println("a=",a)   // a= [1 4 5]
fmt.Println("c=",c)   // c= [1 4 5]

c = append(c, 6)        // forces allocation!
fmt.Println("a=",a)   // a= [1 4 5]
fmt.Println("c=",c)   // c= [1 4 5 6]

c[0] = 9                // mutates a different array!
fmt.Println("a=",a)   // a= [1 4 5]
fmt.Println("c=",c)   // c= [9 4 5 6]
```

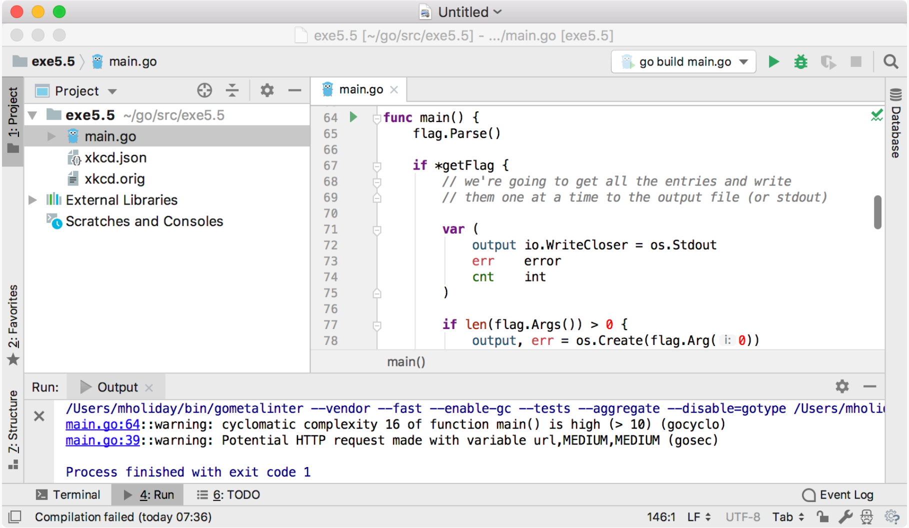
Development Environments

Vim



Vim setup example

Jetbrains GoLand IDE



Packages

Everything lives in a package

Every standalone program has a `main` package

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

Nothing is “global”; it’s either in your package or in another

It’s either at **package** scope or **function** scope

Packages control visibility

Every name that's **capitalized** is exported

```
package secrets

import . . .

type internal struct {
    . . .
}

func GetAll(space, name string) (map[string]string, error) {
    . . .
}
```

That means another package in the program can import it

Within a package, *everything* is visible even across files

Package-level declarations

You can declare anything at *package* scope

```go
package secrets

const DefaultUUID = "00000000-0000-0000-0000-000000000000"

type k8secret struct {
    . . .
}

var secretKey string

func Do(it string) error {
   . . .
}
```

But you can't use the short declaration operator `:=`

Imports

Each *source file* in your package must import what it needs

```
package secrets

import (
    "encoding/base64"
    "encoding/json"
    "fmt"
    "os"
    "strings"
)

. . .
```

It may only import what it needs; unused imports are an error

Generally, files of the same package live together in a directory

What makes a good package?

A package should embed deep functionality behind a simple API

```
package os

func Create(name string) (*File, error)
func Open(name string) (*File, error)

func (f *File) Read(b []byte) (n int, err error)
func (f *File) Write(b []byte) (n int, err error)
func (f *File) Close() error
```

The Unix file API is perhaps the best example of this model

Roughly five functions hide a lot of complexity from the user

No cycles

A package “A” cannot import a package that imports A

```
package A

import "B"

//-------------------

package B

import "A"    // WRONG
```

Move common dependencies to a third package

Or eliminate them

Initialization

Items within a package get initialized before `main`

```go
const A = 1

var B int = C
var C int = A

func Do() error {
    . . .
}

func init() {
    . . .
}
```

Only the runtime can call `init`, also before `main`

Functions

Functions in Go

Functions are “first class” objects; you can:

- Define them — even inside another function
- Create anonymous *function literals*
- Pass them as function parameters / return values
- Store them in variables
- Store them in slices and maps (but not as keys)
- Store them as fields of a structure type
- Send and receive them in channels
- Write methods against a function type
- Compare a function var against `nil`

Function scope

Almost anything can be defined inside a function

```
func Do() error {
    const a = 21

    type b struct {
        . . .
    }

    var c int

    func reallyDoIt() {
        . . .
    }
}
```

Methods cannot be defined in a function (only at package scope)

What is scope?

Scope is a term used to denote a region of the program

It's the region of *visibility* of a name

Scopes can be nested:

- Function within package
- Function within function
- Code block within function

Scope

```
package xyz

var a int

func doIt() {
    var b int
    a = 2

    if b < 10 {
        a := 10
        . . .
    }
}
```

Package-level `a` can be seen inside `doit`, but `b` is *local* to `doit`

There's another `a` inside the `if` block — it *shadows* `xyz.a`

Scope vs lifetime

Scope is static, based on the code at compile time

Lifetime depends on program execution

```
package xyz

func doIt() *int {
    var b int
    . . .

    return &b
}
```

`b` can only be seen inside `doit`, but it will live past the return

It will live so long as part of the program keeps a pointer to it

Shadowing short declarations

Short declarations with `:=` have some gotchas

```
func main() {
    n, err := fmt.Println("Hello, playground")

    if _, err := fmt.Println(n); err != nil {
        fmt.Println(err)
    }
}
```

Compile error: the first `err` is unused

This follows from the scoping rules, because `:=` is a declaration and the second `err` is in the scope of the `if` statement

Shadowing short declarations

Short declarations with `:=` have some gotchas

```
func BadRead(f *os.File, buf []byte) error {
    var err error

    for {
        n, err := f.Read(buf) // shadows 'err' above

        if err != nil {
            break // causes return of wrong value
        }

        foo(buf)
    }

    return err // will always be nil
}
```

Function signatures

The *signature* of a function is the order & type of its parameters and return values

It does not depend on the names of those parameters or returns

```
var try func(int, int) int

func Do(a, b int) int {
    . . .
}

func NotDo(x int, y int) a int {}
    . . .
}
```

These functions have the same *structural* type

Structural typing

It’s the same type if it has the same structure or behavior

```go
type x func(int) int

func main() {
    var a x      // x is a named type

    b := func(y int) int {
        return y+2
    }

    a = b        // b is an anon func, but compatible
    fmt.Println(a(12))
}
```

Go does use *structural* typing in most cases

Structural typing

It's the same type if it has the same structure or behavior:

- arrays of the same size and base type
- slices with the same base type
- maps of the same key and value types
- structs with the same sequence of field names/types
- functions with the same parameter & return types

Named typing

It's the only the same type if it has the same defined name

```
type x int

func main() {
    var a x       // x is a defined type; base int

    b := 12       // b defaults to int

    a = b         // TYPE MISMATCH

    a = 12        // OK, untyped literal
}
```

Go does use *named* typing for non-function *defined* types

Parameter passing

Parameters may be passed *by value* or *by reference*

```
func do(b []int) int {
    b[0] = 0
    b = []int{5, 6, 7}
    return b[2]
}

func main() {
    a := []int{1, 2, 3}
    v := do(a)

    fmt.Println(a, v)    // [0,2,3] 7
}
```

"By value" — the parameter is copied into the function

"By reference" — the function can change the actual parameter

Parameter passing

By value:

- numbers
- bool
- arrays
- structs

By reference:

- pointers to things, including structs
- strings (but they're immutable)
- slices (actually, a reference to the backing array)
- maps
- channels

Return values

Functions can have multiple return values

```go
func doIt(a int, b []int) int {
    . . .

    return 1
}

func doItAgain(a string) (int, error) {
    . . .

    return 1, nil
}
```

Every return statement must have all the values specified

Recursion

A function may call itself; the trick is knowing when to stop

```go
func walk(node *tree.T) int {
    if node == nil {
        return 0
    }

    return node.value + walk(node.left) + walk(node.right)
}
```

This works because each function call adds context to the stack and unwinds it when done

If you don’t have good stopping criteria, the program will crash

Closures

What is a closure?

A *closure* is when a function inside another function “closes over” one or more local variables of the outer function

```
func fib() func() int {
    a, b := 0, 1

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

The inner function gets a **reference** to the outer function’s vars

Those variables may end up with a much longer *lifetime* than expected — as long as there’s a reference to the inner function

Closures: scope vs lifetime

The inner variables continue to live on

```
func fib() func() int {
    a, b := 0, 1

    // return a closure over a & b
}

func main() {
    f := fib()

    // f keeps ahold of a and b and updates them

    fmt.Println(f(), f(), f(), f(), f(), f())
}
```

The inner function continues to mutate the variables it references

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```go
func main() {
    s := make([]func(), 4)

    for i := 0; i < 4; i++ {
        s[i] = func() {
            // they all point to the same "i"
            fmt.Printf("%d %p\n", i, &i)
        }
    }

    for i := 0; i < 4; i++ {
        s[i]()
    }
}
```

The program prints 4 each time; addresses all the same

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```go
func main() {
    s := make([]func(), 4)

    for i := 0; i < 4; i++ {
        j := i // capture it before the closure
        s[i] = func() {
            fmt.Printf("%d %p\n", j, &j)
        }
    }

    for i := 0; i < 4; i++ {
        s[i]()
    }
}
```

The program prints 1, 2, 3, 4 as expected; addresses different

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```
func main() {
    s := make([]func(), 4)

    for i := 0; i < 4; i++ {
        i := i // capture it before the closure
        s[i] = func() {
            fmt.Printf("%d %p\n", i, &i)
        }
    }

    for i := 0; i < 4; i++ {
        s[i]()
    }
}
```

This does the same thing; one `i` shadows the other

Defer

Deferred execution

How do we make sure something gets done?

- close a file we opened
- close a socket / HTTP request we made
- unlock a mutex we locked
- make sure something gets saved before we're done
- ...

The `defer` statement captures a function *call* to run later

Defer

We need to ensure the file closes no matter what

```
func main() {
    f, err := os.Open("my_file.txt")

    if err != nil {
        . . .
    }

    defer f.Close()

    // and do something with the file
}
```

The call to `Close` is guaranteed to run at function exit

Don't defer closing the file until we know it really opened

Defer gotcha #1

The scope of a `defer` statement is the *function*

```
func main() {
    for i := 1; i < len(os.Args); i++ {
        f, err := os.Open(os.Arg(i))

        . . .

        defer f.Close()

        . . .
    }
}
```

The deferred calls to `Close` must wait until function exit

We might run out of file descriptors before that!

Defer gotcha #2

Unlike a closure, `defer` copies arguments to the deferred call

```go
func main() {
    a := 10

    defer fmt.Println(a)

    a = 11

    fmt.Println(a)
}

// prints 11, 10
```

The parameter `a` gets copied at the `defer` statement

The `defer` statement doesn't get a reference

Defer gotcha #2

A `defer` statement runs before the `return` is done

```
func doIt() (a int) {
    defer func() {
        a = 2
    }()

    a = 1
    return
}

// returns 2
```

We have a named return value and a “naked” return

The deferred anonymous function can update that variable

Homework

Homework #2

Exercise 5.5 from *GOPL:* implement `countWordsAndImages`

Actually, given some HTML as raw text, parse it into a document and then call your counting routine to detect and count words and images (you can follow the book's example).

Don't worry about getting HTML from an HTTP query; we're not there yet.

See Homework #1 for counting words.

What happens if the HTML document is empty?