Go book club

•••

Chapter one: Tutorial

Origins of Go

Started at Google in 2007 by Rob Pike, Ken Thompson and Robert Griesemer, open sourced in 2009, version 1.0 released in 2012.

The language emerged because the designers were fed up with the issues of then current systems programming languages. They wanted to create a language that resolved such issues.

Hello world

Packages and Imports: Helps with scoping and visibility

Package level declarations: functions, vars, consts

Main package and func: Special, executable

No semicolons

Build with `go build` -> binary

Useful tools: go fmt, goimports

```
// go get gopl.io/ch1/helloworld

package main

import "fmt"

func main() {

    fmt.Println("Hello, 世界")

}
```

Echo One

Grouping imports, bars, constants

Multi-initialization with zero values:
“” for string, 0 for int, etc.

For loop: no parenthesis, semicolon needed

i++ is a statement, not expression
i.e. “a = i++” => compile error

String concat just like in Java.

For: init and post is optional.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
       s += sep + os.Args[i]
       sep = " "
    }
    fmt.Println(s)
}
```

```
Valid for loops:

for init; cond; post {...}

for init; cond {...}

for cond {...} //while(cond)

for {...} //while(true)
```

Echo Two and Three

Slices: dynamically resizing zero-indexed arrays (e.g. java.util.ArrayList)

Get range of elements: slice[0:5] -> returns 0,1,2,3,4

[10:], [:10] -> 10 to end, start to 9

Works on strings: s[10:] is a substring op

Blank identifier: _ -> Compile error if var declared but not used (same for imports)

Rich stdlib

```
// Package, imports omitted

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
       s += sep + arg
       sep = " "
    }
    fmt.Println(s)
}

// or:

strings.Join(os.Args[1:], " ")
```

```
Valid for loops:

for init; cond; post {...}

for init; cond {...}

for cond {...} //while(cond)

for {...} //while(true)

for index := range slice {}
// same as for loop until
end of array

for i, e := range slice {}
// foreach with indexes
```

Exercises

1.1 and 1.2 are trivial

1.3: Benchmark to the rescue - to be discussed @ chapter 11.

For now, on the right:
Name of benchmark, # of times it ran, time it took for each iteration to complete.

```
// 3 arguments
BenchmarkEchoTwoShort-8            10000000      223 ns/op
BenchmarkEchoThreeShort-8          10000000      184 ns/op

// 192 arguments
BenchmarkEchoTwoLong-8             50000         34531 ns/op
BenchmarkEchoThreeLong-8           500000        2605  ns/op

// 768 arguments
BenchmarkEchoTwoLongExtra-8        3000          402806 ns/op
BenchmarkEchoThreeLongExtra-8      200000        9371   ns/op
```

Duplicates

Note: counts never printed

map[string]int is like *HashMap<String, Integer>*, getting-setting is done via
val := counts[key]
counts[key] = val

Key can be anything that has "=="

The zero value for map is "nil". To get an initialized empty map, "make" is used.

Getting a value from map that isn't there returns 0 value

Print formats

```go
func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
       counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
    for line, n := range counts {
       if n > 1 {
             fmt.Printf("%d\t%s\n", n, line)
       }
    }
}
```

Duplicates Two

Multi-returns: os.Open returns file handle and an error. If err is not nil, opening the file failed, otherwise all good.

Pattern repeated everywhere where an operation can fail.

Opened files (and other resources) must be closed after use.

```go
func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
}
```

Duplicates Three

ioutil.ReadFile reads the file into the data variable ([]byte)

Casting is done via type() calls: int(13.0), float64(255), string(someByteArray), etc.

Note: Carriage return (on Windows) not handled.

```
func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
                fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
                continue
        }
        for _, line := range strings.Split(string(data), "\n") {
                counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
                fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Exercise 1.4

Modify **dup2** to print the names of all files in which each duplicated line occurs.

The GIFs

const - fixed at compile time (static final), must be number, string or boolean. Can be pkg or func lvl

```
const whiteIndex = 0
```

using imported package by last entry in path

```
import “math/rand”
freq := rand.Float64() * 3.0
```

struct: group of values (*fields*) making up an object that can be treated as a unit

```
type RGBA struct {
    R, G, B, A uint8
}
```

composite literal: instantiate structs, maps and slices with elements. Works with and without naming fields

```
var palette = [ ]color.Color{color.White,
color.Black}
anim := gif.GIF{LoopCount: nframes}
rgba := RGBA{1, 2, 3, 4}
```

Exercise 1.5 and 1.6

Modify **lissajous** to create green-on-black gifs

Modify **lissajous** to produce images in multiple colors

Basic Fetch

Multi-return, check for errors

Closing the response body

Exercise 1.7: use io.Copy(dst, src) instead of ioutil.ReadAll()

Exercise 1.8: Add protocol prefix if missing (strings.HasPrefix())

Exercise 1.9: Also print the HTTP status code

```
func main() {
    for _, url := range os.Args[1:] {
       resp, err := http.Get(url)
       if err != nil {
             fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
             os.Exit(1)
       }
       b, err := ioutil.ReadAll(resp.Body)
       resp.Body.Close()
       if err != nil {
             fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url,
err)
             os.Exit(1)
       }
       fmt.Printf("%s", b)
    }
}
```

A web server

First class functions: Function can also be passed in parameters, returned from functions and assigned to variables

Mutual Exclusion locks: Basically synchronized(obj) {...} block. Goroutines can only *Lock()* a Mutex if it is not currently locked. Used to avoid race conditions.

```go
var mu sync.Mutex
var count int

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000",
nil))
}
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}
```

Another web server

value init in if statement: if allows initialization of a variable in its evaluation. Same as if you did the initialization on the previous line.

```go
func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe("localhost:8000",
nil))
}
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL,
r.Proto)
    for k, v := range r.Header {
       fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil {
       log.Print(err)
    }
    for k, v := range r.Form {
       fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}
```

Exercise 1.12

Modify the **lissajous** server to read parameter values from the URL

Loose ends - control flows

Switch: no fallthrough by default, can evaluate expressions both in operand and in cases.

Continue and ***break***: Same as in Java

Goto: Yes, there's goto.

Loose ends - pointers

Pointer is a value that contain the address of a variable. Somewhat constrained, no pointer arithmetic.

& operator yields the address of variable, * operator retrieves the variable that the pointer refers to, e.g. &addr and *p

By default, Go passes around everything by copying (like primitives in Java), not as a reference (like Objects in Java). Passing around pointers is basically the same as pass-by-reference in Java.

Loose ends - methods and interfaces

Named type: var p Point <- p is a named type of Point

Method: function associated with a named type:

```
func (p Point) travel() {…}
p.travel()
```

Interface: abstract type that allows us to treat different concrete types in the same way based on what methods they have. Interfaces are inherited automatically. Named type ‘p’ implements the ‘Traveler’ interface because it has the ‘travel()’ method.

```
type Traveler interface {
    travel()
}
```

Loose ends - comments

Pretty much the same as in Java, but:

1. Comment before package declaration is used as documentation for the package
2. Comment before exported function/method is used as documentation for function/method

See go doc

Next session

Chapter 2 (?)