Go Language

August 2015

Giacomo Tartari
PhD student, University of Tromsø

Go lecture 1/2

Introduction

Motivation

Syntax

Capabilities

Example code

Go lectures 2/2

Practicalities

Installation

Environment

Tooling

HowTos

Demo?

Go

A new language?

Why?

and what for?

Don't we have Java?

C?

C++?

C#?

D?

Haskell?

Scala?

Python?

PHP?

Ruby?

Perl?!

Brainfuck!!!!

[add random language here]?

A new language

What's wrong with all of the above?

Rob Pike's take (one of the Go instigator) (http://talks.golang.org/2012/splash.article)

Languages used at Google were not satisfactory

- These languages were developed before the multi-core revolution
- Millions of lines of code maintained by thousands of programmers
- Build times of many minutes or hours

Go

Modern and pragmatical language

Not a research language to explore new horizons

A language to get the job done

Designed by and for people who build and maintain large systems

Easy to read, clean syntax

Good tools

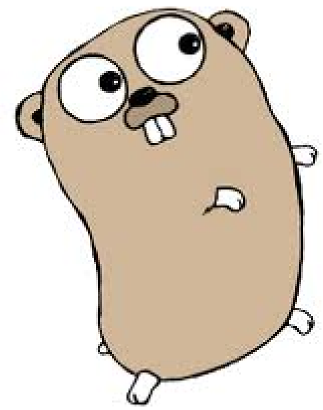
Nothing exactly new but a collection of good features

Go

Opensourced in 2009

Current version 1.5

Stable since 2012: Go 1 promise



Go

C-like syntax

Compiled to machine code

CSP-like Concurrency

Garbage collected

Static and strong typed

No exceptions for handling errors

No inheritance but composition

No generics

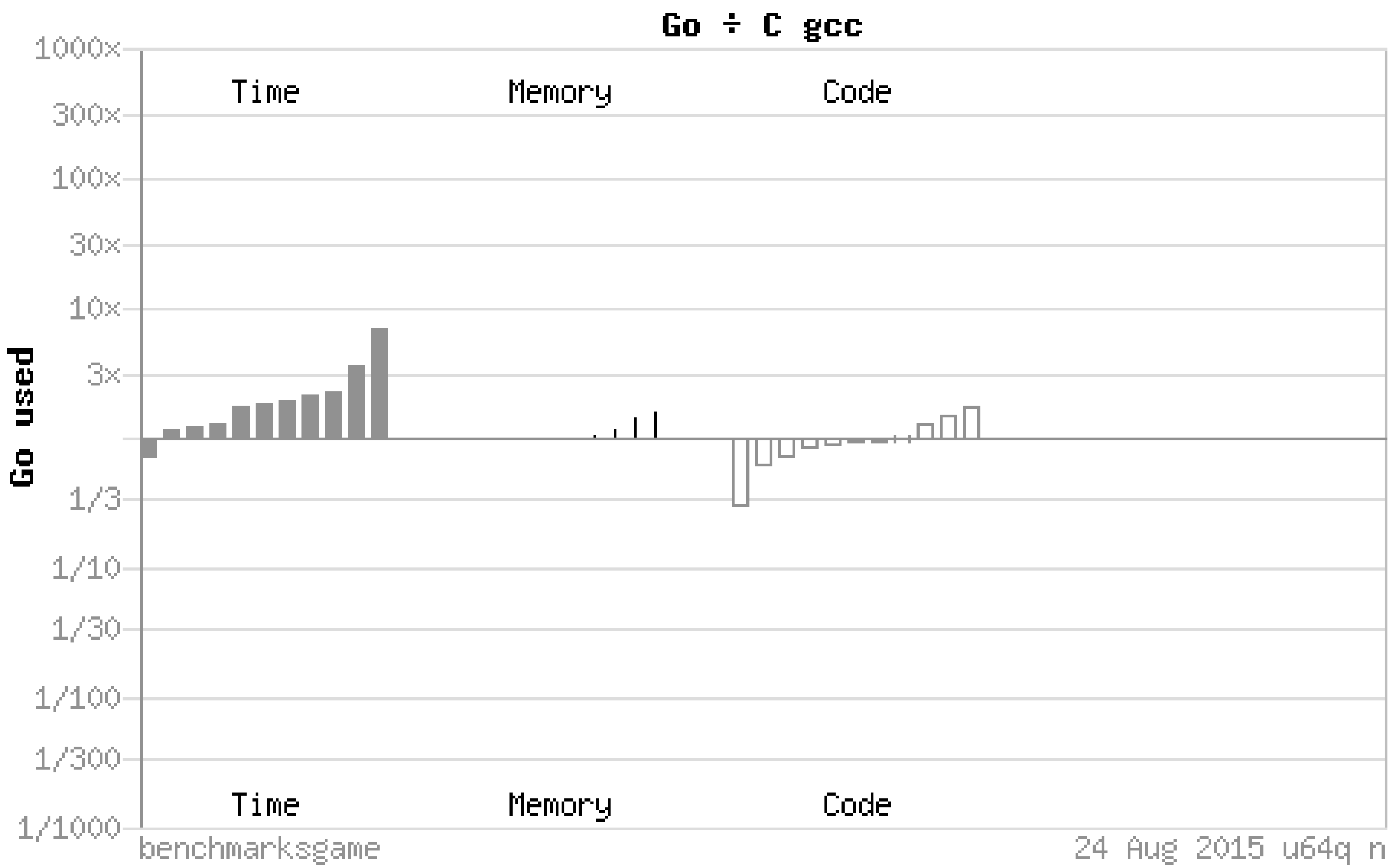
No header files

golang.org/doc/faq (http://golang.org/doc/faq)

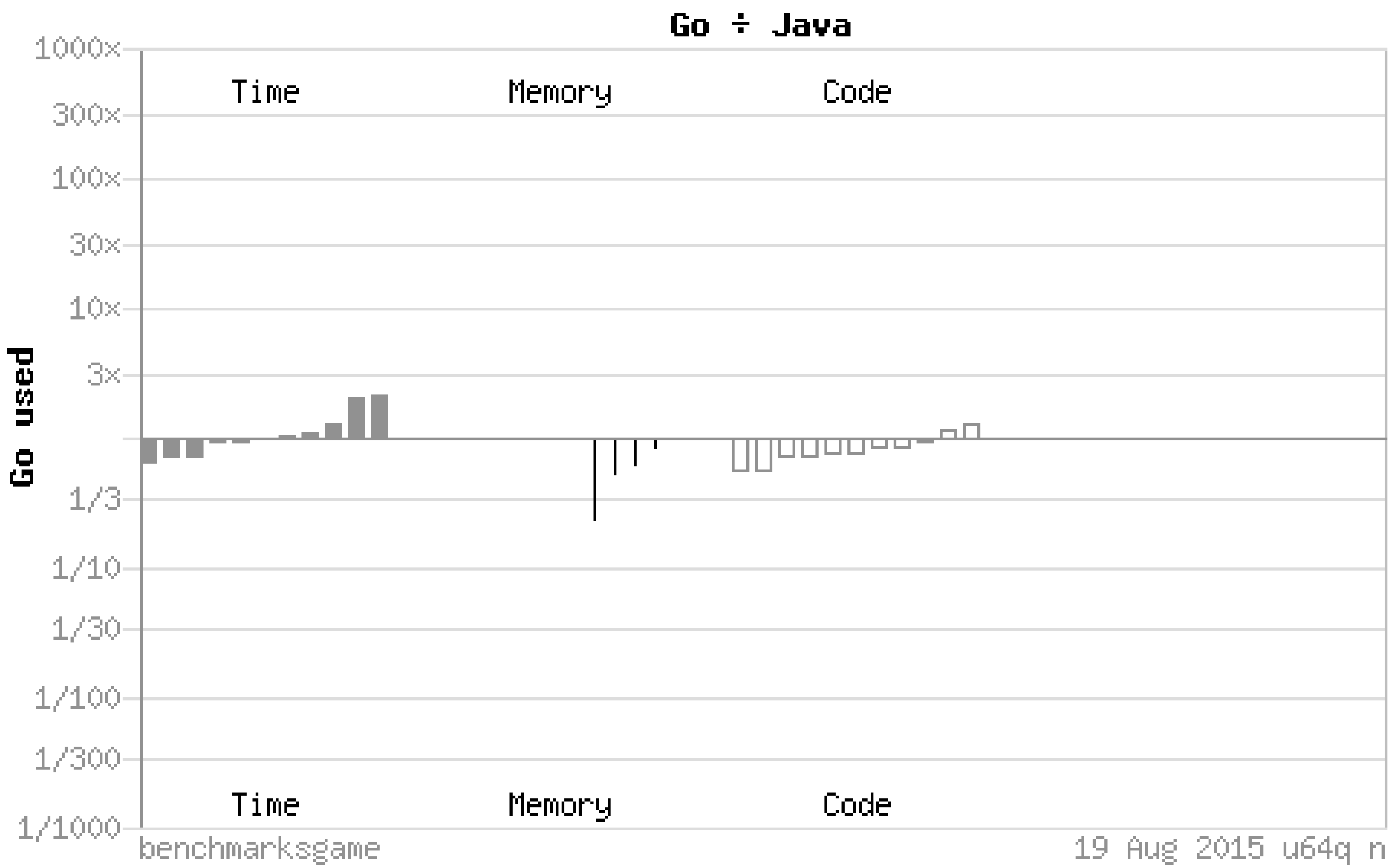
Go benchmarks (!)

Benchmarks from here (http://benchmarksgame.alioth.debian.org/u64q/go.php)

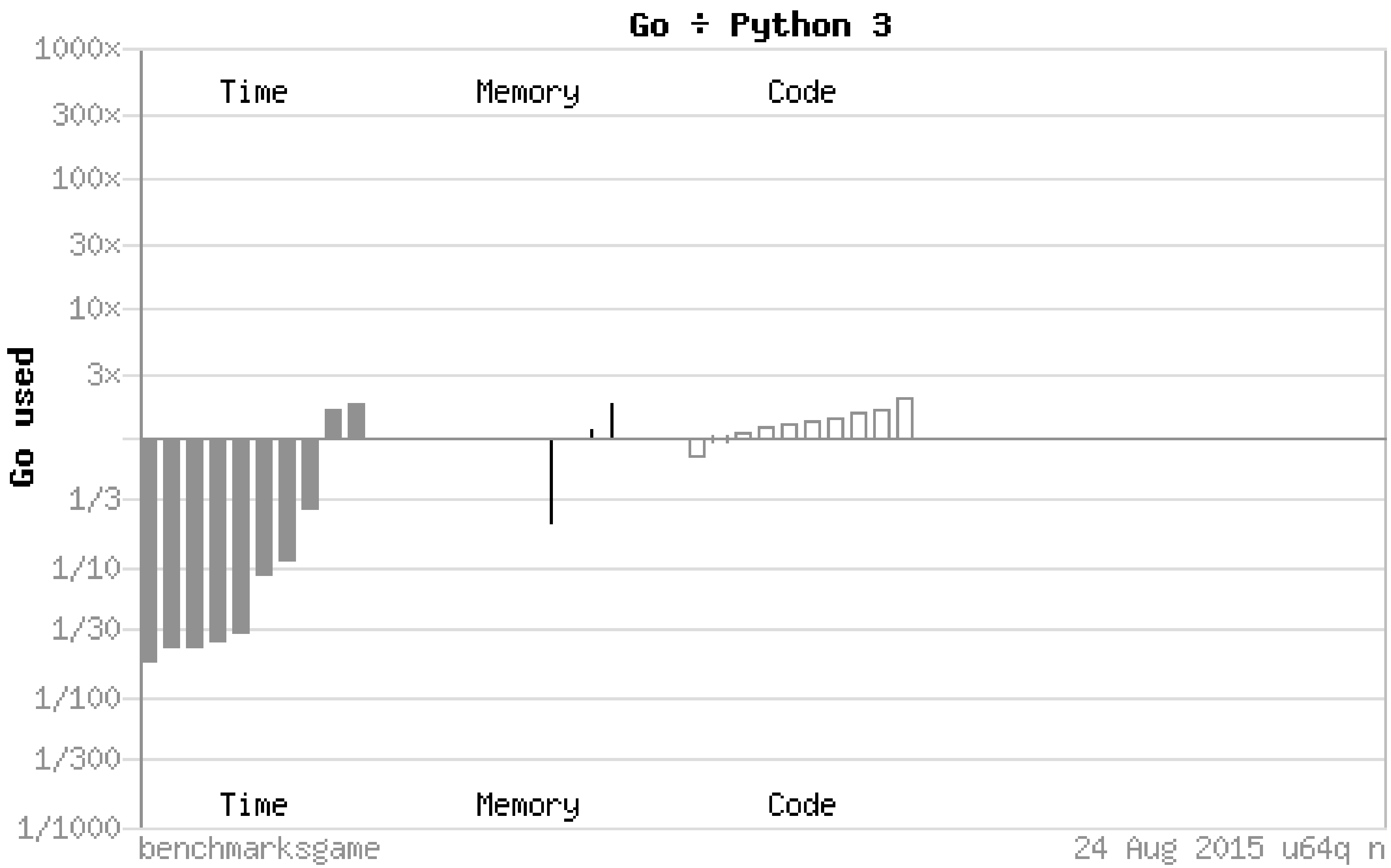
Go benchmarks (!)



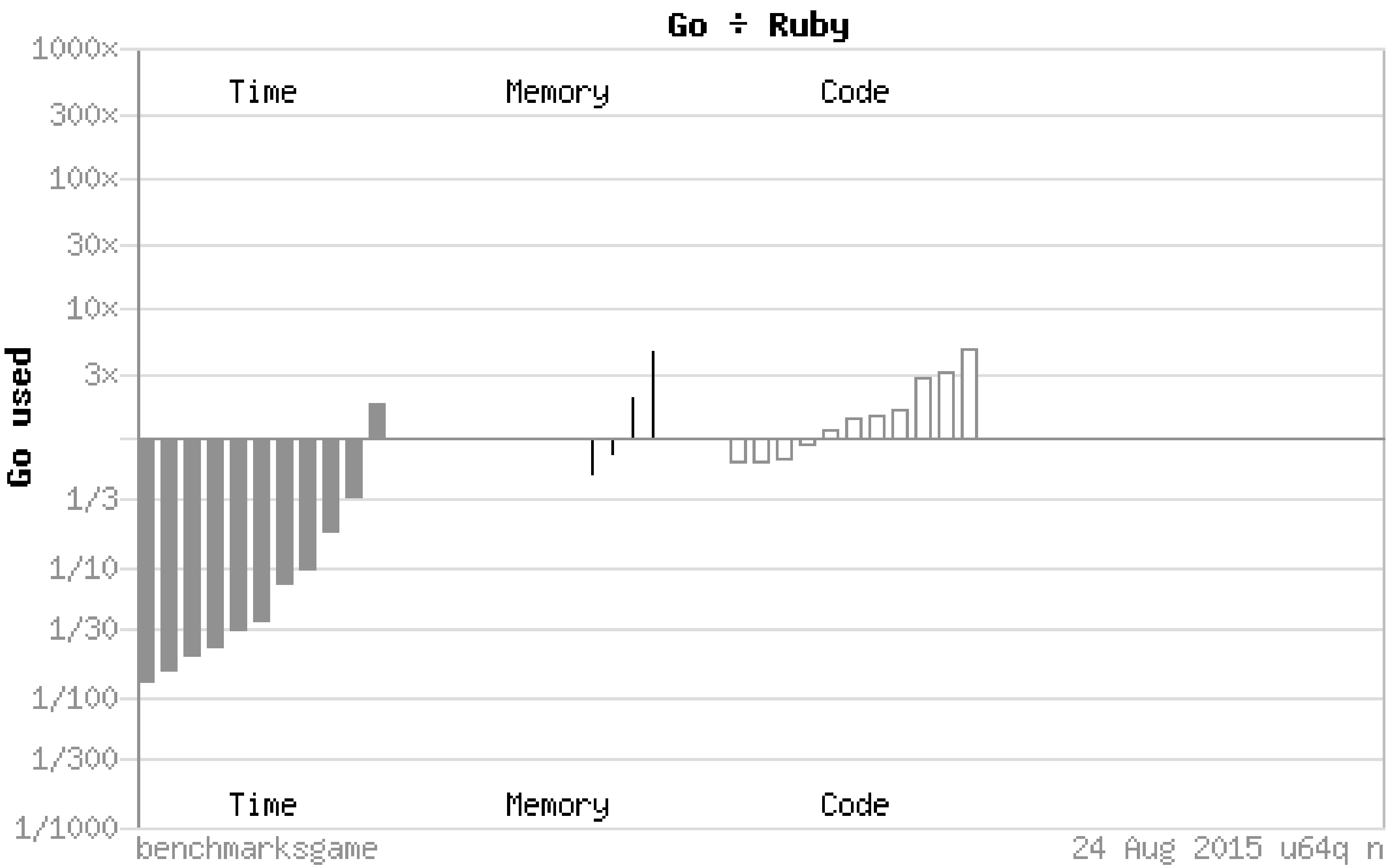
Go benchmarks (!)



Go benchmarks (!)



Go benchmarks (!)



Hello World

```
package main

import "fmt"

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

Run

Basic types

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32
     // represents a Unicode code point

float32 float64

complex64 complex128
```

Also array, slice, maps and channels

But we'll see them later

Some operators

```
+    sum                    integers, floats, complex values, strings
-    difference             integers, floats, complex values
*    product                integers, floats, complex values
/    quotient               integers, floats, complex values
%    remainder              integers

&&    conditional AND    p && q  is  "if p then q else false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"

==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal

&x    address operator
*p    pointer indirection
```

No pointer arithmetics

Packages

- Go packages mix the properties of libraries, name spaces, and modules
- A package is compiled in a static library or in a (statically linked) executable if `main.main()` is present
- Multiple files can be part of a package
- No restriction to what can be in a file, but the files must be in the same dir
- Name visibility outside packages is a *property of the name*

Packages, exported identifiers

```
package mypackage

import (
    "errors"
    "fmt"
    "github.com/user/package"
)

var A int //exported
func MyFunc(){...} //exported
var b float32 //not exported
```

A name is visible outside its package iff

- The first character of the identifier's name is upper case
- The identifier is declared in the package block or it is a field name or method name.

Remember `fmt.Println(...)` in hello world?

Variables and Constants

```
package test

var (
    B       string = "hello"
    x, y, z float32
    p       *int
)

const (
    C   = iota //0
    D          //1
    E          //2
)
```

As imports variables and constants can be declared in blocks

Variables, and constants, can be initialized when declared

Together with the iota constant generator it permits light-weight declaration of sequential values

More variables

Inside functions variable can be defined with `:=`

```
x := SomeFunc()
y := x++
x, y, z := 0, 1, 2
```

The compiler infers the type

N.B. In multiple assignment at least one receiving variable must be declared

```
var x = DoSomething() //OK

err := DoStuff() //OK short declaration and assignment

a, err := DoMoreStuff() //OK a is declared here

a, err = DoOtherStuff() //NOT OK both vars have already been declared
```

Functions

- First class functions
- Higher order functions
- User defined function types
- Function literals
- Closures
- Multiple return values

Functions

First-class functions, higher-order functions and user-defined function types

```
package main

import "fmt"

func AddOne(val int) int {
    return val + 1
}

func AddMore(fn func(int) int) int {
    return fn(41)
}

func main() {
    f := AddOne
    x := 41
    fmt.Println("The answer is", f(x))
    fmt.Println("The answer is", AddMore(f))

}
```

Run

More functions

Function literals are closures

```
package main

import "fmt"

func main() {
    x := 5
    f := func() int {
        x++
        return x
    }
    g := func(y int) {
        x = y
    }
    fmt.Println("x:", x)
    f()
    fmt.Println("x after f():", x)
    g(42)
    fmt.Println("x after g(42):", x)
}
```

Run

Deferred functions

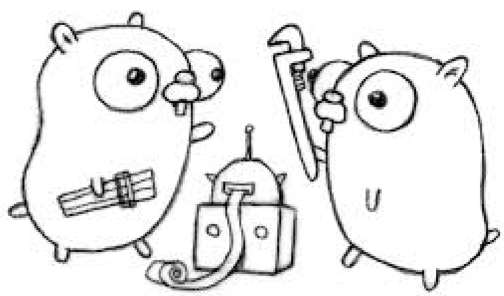
A defer statement schedules a function call to be run just before the function executing the defer returns

The canonical examples are unlocking a mutex or closing a file

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fname := "/tmp/file.tmp"
    f, err := os.Open(fname)
    if err != nil {
        /*handle error*/
    }
    defer f.Close()

    // do stuff with file
}
```

Types, allocations and composition

- Type definition
- Method definition
- Allocation
- Interfaces
- Composition and embedding



Type definition

```
type MyInt int

type Vertex struct{
    X, Y int
}

type Writer interface{
    Write()
}

type Callback func(*int, string)
```

Go is strong typed so `MyInt` is not an `int`

Types can be converted with this syntax `int(MyInt)`

Any type can have methods (even functions)

Allocation

Struct literals

```
var (
    p = Vertex{1, 2}  // has type Vertex
    q = &Vertex{1, 2} // has type *Vertex
    r = Vertex{X: 1}  // Y:0 is implicit
    s = Vertex{}      // X:0 and Y:0
    t = Vertex{X: 3, Y: 4}
)
```

Built in function `new()` returns a pointer to a newly allocated and zeroed memory

```
v := new(Vertex) // has type *Vertex
```

Built in function `make()` used to allocate built in types: channels, maps and slices

```
m := make(map[string]int)
```

Method

```
func (v1 Vertex)Add(v2 Vertex)Vertex{
    return Vertex{v1.X + v2.X, v1.Y + v2.Y}
}

func (v *Vertex)Sum(a Vertex) {
    v.X += a.X
    v.Y += a.Y
}
```

Method receivers and method sets

Vertex

- Add(Vertex)

*Vertex

- Add(Vertex)
- Sum(Vertex)

Method receivers

```
type Vertex struct {
    X, Y int
}

func (v1 Vertex) Add(v2 Vertex) Vertex {
    return Vertex{v1.X + v2.X, v1.Y + v2.Y}
}

func (v *Vertex) Sum(a Vertex) {
    v.X += a.X
    v.Y += a.Y
}

func main() {
    a := Vertex{1, 3}
    b := Vertex{2, 6}
    fmt.Println(a)
    a.Sum(b)
    fmt.Println(a)
}
```

Run

If x is addressable and &x's method set contains m, x.m() is shorthand for (&x).m()

Interfaces

Interfaces are a sets of methods

Just behavior

Often just a few methods

From pkg/io

```
type Writer interface {
    Write(p []byte) (n int, err error)
}

type Reader interface {
        Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

Interfaces

An interface is satisfied if the type implements all the methods in the set

No *implements* keyword

Interface satisfaction is statically checked at compile time

Interfaces are type-safe

Structural typing, is like duck typing but better

The compiler tells you if it is a *duck*

Interfaces

Any type satisfies the empty method set

```
interface{}
```

```
package main

import "fmt"

func main() {
    var i interface{}
    x := 127
    i = x
    fmt.Println("i:", i)
    v := struct{ π, e float32 }{3.14159, 2.71828}
    i = v
    fmt.Println("i:", i)
}
```

Run

Method sets

N.B.

```
type Worker interface {
    DoStuff()
}

type Mule struct{}

func (m *Mule) DoStuff() {
    fmt.Println("It's hard work!!")
}

func main() {
    m := Mule{}
    var w Worker
    w = m
    w.DoStuff()
}
```

Run

The concrete value stored in an interface is not addressable

Interfaces composition

Interfaces can be composed

From pkg/io

```
type ReadWriteCloser interface {
        Reader
        Writer
        Closer
}

func MyFunc(stream io.ReadWriteCloser) error{
    ...
    stream.Read()
    sream.Write(data)
    stream.Close()
    ...
}
```

Type assertion

```
type error interface {
        Error() string
}
```

```
type PathError struct{
    Path string
    Ctx *Context
    Timestamp time.Time
}

func (pe PathError) Error() string{
    return fmt.Sprintf("Wrong path: %s", pe.Path)
}
```

```
err := FuncPath(...)
ep, ok := err.(PathError)
if ok{
    ep.Ctx
}
```

Struct embedding

Also struct can be composed by embedding

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

type User struct {
    Person
    Id int
}

func main() {
    u := User{}
    u.Name = "Adam" // u.Person.Name = "Adam"
    u.Age = 42      //u.Person.Age = 42
    u.Id = 1
    fmt.Println(u)
}
```

Run

Struct embedding

And the interface they are satisfying as well

```
type Locker interface {
        Lock()
        Unlock()
}
```

```
import "sync"

type Vertex struct {
    X, Y int
}

type VertexLocker struct{
    sync.Mutex
    Vertex
}

vl := VertexLocker{}
vl.Lock()
vl.X = 99
vl.Unlock()
```

Flow Control: if, for and switch

If

```
package main

import (
    "fmt"
    "math"
)

func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}

func main() {
    fmt.Println(
        pow(3, 2, 10),
        pow(3, 3, 20),
    )
}
```

Run

For

```
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum)
}
```

Run

As in C or Java, you can leave the pre and post statements empty

And drop the semicolons: C's while is spelled for in Go

Switch

Not just numbers

```
package main

import "fmt"

func main(){
    switch s := "yes"; s{
    case "yes":
        fmt.Println("yes")
    case "no", "noway":
        fmt.Println("no")
    case "maybe":
        fmt.Println("maybe")
    default:
        fmt.Println("whatever")
    }
}
```

Run

Type switch

What is the actual type of an interface?

```
err := json.Unmarshal(data, &p)
if err != nil {
    switch t := err.(type) {
    case *json.UnmarshalFieldError:
        log.Println(t)
    case *json.UnmarshalTypeError:
        log.Println(t)
    case *json.UnsupportedTypeError:
        log.Println(t)
    case *json.UnsupportedValueError:
        log.Println(t)
    case *json.SyntaxError:
        log.Println(t)
    case *json.InvalidUnmarshalError:
        log.Println(t)
    }
    return err
}
```

Arrays, slices and maps

Arrays

```
package main

import "fmt"

var b [10]int
var f [3]*float64

func main() {
    primes := [...]int{2, 3, 5, 7, 11, 13, 17}

    fmt.Println("primes", primes)

    fmt.Println("b", b)

    fmt.Println("f", f)

}
```

Run

Arrays

Value type not reference type

The size of an array is part of its type

```
package main

import "fmt"

var a, b [4]int
var c [3]int

func main() {
    a[0], a[1] = 1, 2
    b = a
    fmt.Printf("a: %v\nb: %v\n\n", a, b)
    a[3] = 3
    fmt.Printf("a: %v\nb: %v\n", a, b)

    // [3]int != [4]int
    //c = a
}
```

Run

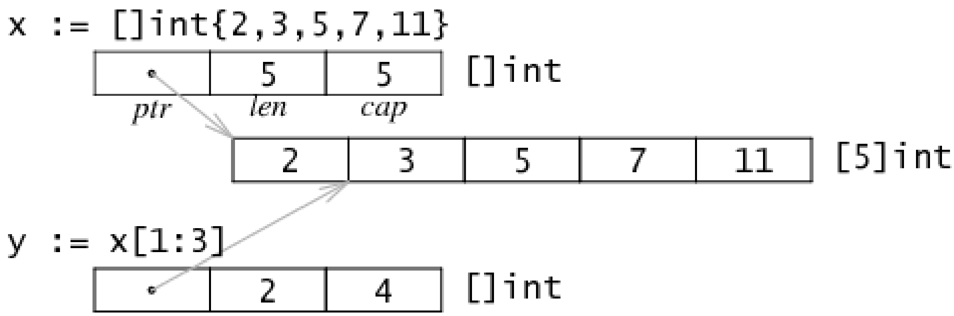
Slices

```
package main

import "fmt"

func main() {

    p := []int{2, 3, 5, 7, 11, 13}

    fmt.Println("p ==", p)
    fmt.Println("p[1:4] ==", p[1:4])

    // missing low index implies 0
    fmt.Println("p[:3] ==", p[:3])

    // missing high index implies len(s)
    fmt.Println("p[4:] ==", p[4:])
}
```

Run

Just a **slice** of an array

Slice internals




```go
package main

import "fmt"

func main() {
    //func make([]T, len, cap) []T
    sl := make([]int, 3, 5)

    fmt.Println(sl)
    fmt.Println(len(sl), cap(sl))

    nsl := append(sl, 3)
    fmt.Println(nsl)
    fmt.Println(len(nsl), cap(nsl))

}
```

Run

Slice built in functions

Append to a slice

```
func append(slice []Type, elems ...Type) []Type
```

Slice capacity

```
func cap(v Type) int
```

Slice length

```
func len(v Type) int
```

Copy a slice

```
func copy(dst, src []Type) int
```

Slice tricks

Append

```
a = append(a, b...)
```

Copy

```
b = make([]T, len(a))
copy(b, a)
```

Cut

```
a = append(a[:i], a[j:]...)
```

Delete

```
a = append(a[:i], a[i+1:]...)
// or
a = a[:i+copy(a[i:], a[i+1:])]
//without preserving order
a[i], a = a[len(a)-1], a[:len(a)-1]
```

etc...

Maps

```
var m = map[string]int{"one": 1, "two": 2, "three": 3}

func main() {
    //m := make(map[string]int)

    m["five"] = 5
    fmt.Println(m)

    delete(m, "three")
    fmt.Println(m)

    b, ok := m["four"]
    if !ok {
        fmt.Println("cannot find four")
    }else{
        fmt.Println(b)
    }

    fmt.Println(m["four"])
}
```

Run

Keys can be integers, floats, complex, strings, pointers, interfaces, structs, arrays

Range

```
package main

import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

var m = map[string]int{"one": 1, "two": 2, "three": 3, "four": 4, "five": 5}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }

    for k, v := range m{
        fmt.Printf("Key: %s, Value: %d\n", k, v)
    }


}
```

Run

Concurrency: goroutines and channels

Concurrency vs Parallelism

concur.rspace.googlecode.com/hg/talk/concur.html#title-slide

(http://concur.rspace.googlecode.com/hg/talk/concur.html#title-slide)

- Concurrency != parallelism
- Concurrency is the composition of independently executing processes
- Concurrency enables parallelism
- Concurrency is about structure
- Parallelism is about execution
- A concurrent program can be executed correctly on one CPU

Goroutine

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Run

Concurrency

Goroutines provide concurrency

- go statement allows us to run functions independently in different goroutines
- Goroutines live in the same address space
- Think of them as a very lightweight threads

Now we need to communicate

- We need return values form the goroutines
- We need to feed fresh data to the goroutines to be elaborated

Communication

Same address space you say?

Synchronize access to shared memory

- Fence some sheared memory with mutex, locks, conditions...
- Communicate by reading/writing this shared memory

We can do better, we have channels!!

Channels

Channels can *connect* goroutines and allow them to communicate

The go runtime will take care of the synchronization details

We just care about sending and receiving data

Don't communicate by sharing memory; share memory by communicating.

Channels

```
ch := make(chan int)     // unbuffered channel
ch := make(chan int, 10) // buffered channel
```

Unbuffered channels combine communication with synchronization

Buffered channel are more like synchronized and type safe FIFO queues

Communication primitives

```
ch <- v                  // Send v to channel ch
v := <-ch                // Receive from ch, and assign value to v
```

Both operation are blocking if the channel is not ready to communicate

Same as Unix pipes

- Read blocks while pipe is empty
- Write blocks while pipe is full

Communication

```
func Ping(ch chan *int) {
    for {
        i := <-ch
        time.Sleep(300 * time.Millisecond)
        *i++
        fmt.Println("Ping", *i)
        ch <- i
    }
}

func Pong(ch chan *int) {
    for {
        i := <-ch
        time.Sleep(300 * time.Millisecond)
        *i++
        fmt.Println("Pong", *i)
        ch <- i
    }
}
```

Communication

```
func main() {
    ch := make(chan *int)
    go Ping(ch)
    go Pong(ch)
    var i int
    ch<-&i
    time.Sleep(5 * time.Second)

}
```

Run

The computation happens in the goroutines

The value is passed back and forth

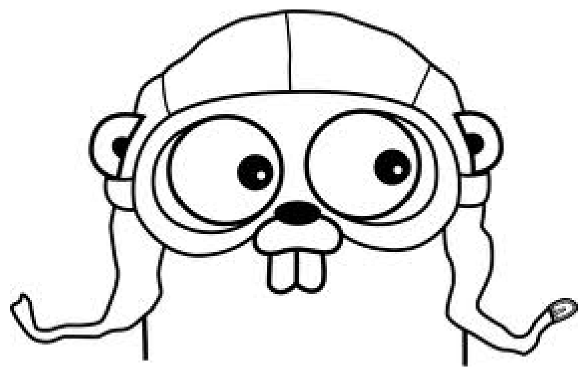
The communication is the synchronization

Sharing is caring

Memory is not fenced by locks and condition

Memory is shared bu *passing* it along

After you give it away is not your memory anymore



More channels

Channels can be closed to signal the receivers the termination of the data flow

```
ch := make(chan int)
close(ch)
```

To check if a channel is closed use the multi-valued assignment form of the receive operator

```
x, ok := <-ch
if !ok{
    fmt.Println("Channel closed!")
}
```

Receiving from a closed channel always succeeds, immediately returning the element type's zero value

Range in channels

```
package main

import "fmt"

func main() {
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)
    for elem := range queue {
        fmt.Println(elem)
    }
}
```

Run

Range will iterate on all the values sent through the channel until it is closed

Closing a non empty channel will not prevent us from receiving the already sent values

What if we do not close the channel?

Select

Select is similar to switch but each case is a communication statement

```
in := make(chan int)
out := make(chan int)

select{
case i := <- in:
    fmt.Println("received i")
case out <- x:
    fmt.Println("sent x")
default:
    fmt.Println("no communication")
}
```

If `default` case is not present `select` blocks until a channel is ready to communicate

`select{}` blocks forever

Select

```
func main() {
    c1 := make(chan string)
    c2 := make(chan string)
    r := rand.New(rand.NewSource(time.Now().UnixNano()))

    go func() {
        time.Sleep(time.Second * time.Duration(r.Intn(3)))
        c1 <- "one"
    }()

    go func() {
        time.Sleep(time.Second * time.Duration(r.Intn(3)))
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

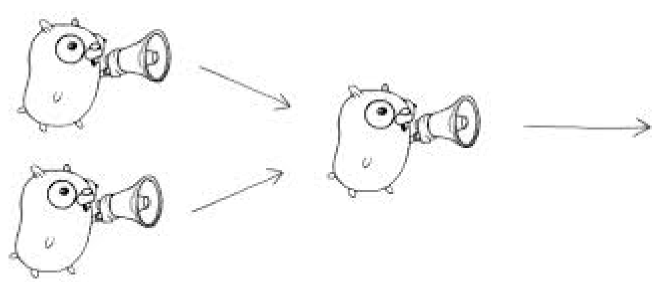
Run

Timeouts

```
timeout := time.After(100 * time.Millisecond)
select {
case result := <-ch:
    DoStuff(result)
case <-timeout:
    fmt.Println("timed out")
    return
}
```

Fan in

```
input1 := make(chan string)
input2 := make(chan string)
out := make(chan string)
for {
    select {
    case s := <-input1:
        out <- s
    case s := <-input2:
        out <- s
    }
}
```



Avoid (some) garbage

Buffered channel can hold resources to be reused

```
var buffers = make(chan *Buffer, 100)

go func(){
    var buff *Buffer
    select{
    case buff = <- buffers:
        //got one
    default:
        buff = new(Buffer)
    }
}
...
go func(){
    select{
    case buffers <- buff:
        //recycle
    default:
        //garbage
    }
}
```

Broadcast a signal

Goroutines are cheap, can have 100000 running on normal hardware

Maybe the goroutines need to cleanup before the application shuts down

Keeping track of how many are alive can be difficult

But receiving from a closed channel always succeeds...

```
var quit = make(chan *struct{})

select{
case buff := <-ch:
    //do stuff
case <- quit:
    //shutdown, close files connections etc...
    //cleanup
    //...
}
```

Generator

```
func idGenerator() chan int {
    ids := make(chan int)
    go func() {
        id := 0
        for {
            ch <- id
            id++
        }
    }()
    return ids
}
...
ids := idGenerator()
id1 := <-ids
id2 := <-ids
```

Practical stuff

Installation

Official binary distributions

golang.org/dl/ (https://golang.org/dl/)

Or from source (requires Go1.4)

Environment

Some optional environment variables

- GOROOT
- GOOS
- GOARCH
- GOBIN

To override the defaults put something like this in your `.bash_profile` or `.profile`

```
export GOROOT=$HOME/go
export GOARCH=amd64
export GOOS=linux
export PATH=$GOROOT/bin:$PATH
```

Environment

One environment variable that is needed is GOPATH

From the help:

```
The Go path is used to resolve import statements.
It is implemented by and documented in the go/build package.

The GOPATH environment variable lists places to look for Go code.
On Unix, the value is a colon-separated string.
On Windows, the value is a semicolon-separated string.
On Plan 9, the value is a list.

GOPATH must be set to get, build and install packages outside the
standard Go tree.
```

e.g. `GOPATH=/home/user/gocode`

GOPATH

More from the help:

```
Here's an example directory layout:

GOPATH=/home/user/gocode

/home/user/gocode/
    src/
        foo/
            bar/               (go code in package bar)
                x.go
            quux/              (go code in package main)
                y.go
    bin/
        quux                   (installed command)
    pkg/
        linux_amd64/
            foo/
                bar.a          (installed package object)
```

Also GOPATH/bin should go in your PATH

Build constraints

A build constraint is a line comment beginning with +*build* that lists the conditions under which a file should be included in the package

```
// +build linux darwin
// +build 386
...
// +build ignore
```

Or if the file name ends in

```
*_GOOS
*_GOARCH
*_GOOS_GOARCH
```

E.g. `source_windows_amd64.go`

go tool(s)

```
$go help
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

    build       compile packages and dependencies
    clean       remove object files
    env         print Go environment information
    fix         run go tool fix on packages
    fmt         run gofmt on package sources
    get         download and install packages and dependencies
    install     compile and install packages and dependencies
    list        list packages
    run         compile and run Go program
    test        test packages
    tool        run specified go tool
    version     print Go version
    vet         run go tool vet on packages
    ...
```

go tool(s)

go get

```
$go get github.com/golang/glog
```

go build

```
$cd $GOPATH/src/myproject
$go build
```

go run

```
$cd $GOPATH/src/myproject
$vim main.go
$go run main.go
```

go fmt (aka end of coding style war!!)

```
$go fmt .
```

godoc

Offline docs

```
$godoc
usage: godoc package [name ...]
godoc -http=:6060
...
```

Online docs for the standard library

golang.org/pkg (http://golang.org/pkg)

Online docs for third party libraries

godoc.org/ (http://godoc.org/)

Online presentation

talks.godoc.org/ (http://talks.godoc.org/)

Integration

$ls -l $GOROOT/misc

```
IntelliJIDEA
arm
bash
bbedit
benchcmp
cgo
chrome
dashboard
dist
emacs
fraise
git
goplay
kate
linkcheck
notepadplus
pprof
swig
vim
xcode
zsh
```

Let's code

How to write Go code

Create a directory

Write the code (and the documentation and the tests)

Have a look at the standard library

Compile it

Run the tests?

See the docs off line?

Tests and benchmark

golang.org/pkg/testing/ (http://golang.org/pkg/testing/)

Put your tests/benchmark in a file ending in `_test.go`

```
import testing

func TestXxx(t *testing.T){
    ...
}

func BenchmarkXxx(b *testing.B){
    ...
}
```

```
$cd $GOPATH/src/mypackage
$go test
$go test -bench=.
```

golang.org/cmd/go/#Description_of_testing_flags (http://golang.org/cmd/go/#Description_of_testing_flags)

Profiling

blog.golang.org/profiling-go-programs (http://blog.golang.org/profiling-go-programs)

github.com/davecheney/profile (http://github.com/davecheney/profile)

Dynamic tools

talks.golang.org/2015/dynamic-tools.slide#1 (https://talks.golang.org/2015/dynamic-tools.slide#1)

Readings

golang.org/ (http://golang.org/)

tour.golang.org/ (http://tour.golang.org/)

gobyexample.com/ (https://gobyexample.com/)

learnxinyminutes.com/docs/go/ (http://learnxinyminutes.com/docs/go/)

talks.golang.org (http://talks.golang.org)

code.google.com/p/go-wiki/w/list (https://code.google.com/p/go-wiki/w/list)

code.google.com/p/go-wiki/wiki/GoTalks (https://code.google.com/p/go-wiki/wiki/GoTalks)

research.swtch.com/godata (http://research.swtch.com/godata)

morsmachine.dk/go-scheduler (http://morsmachine.dk/go-scheduler)

CSP (http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf)

Thank you

Giacomo Tartari
PhD student, University of Tromsø
giacomo.tartari@uit.no (mailto:giacomo.tartari@uit.no)