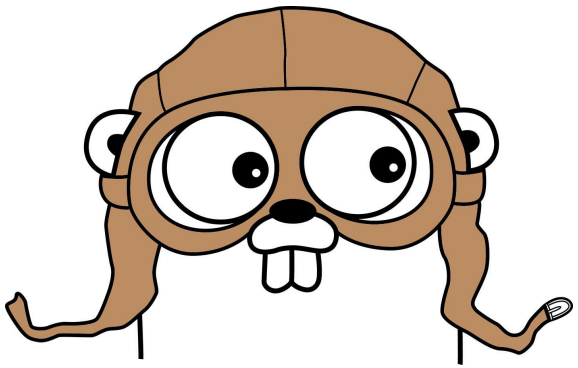
10 Reasons to be Excited About Go

Dvir Volk, Chief Architect,
Everything.me



O HAI! I CAN HAS A GO?

What is Go

- New-ish programming language developed at Google (and used there)
- Targeted for high performance servers
- Focuses on easing the life of the developer
- Addresses real pains - mainly from C++
- Object-Oriented-ish
- Statically Compiled
- Garbage Collected
- Strictly Typed
- Concurrency Oriented

Reason 1: The syntax

- Simple, Quick learning curve, little code to write.
- Few keywords - e.g.: ~~do, while~~, for
- Multiple return values
- Public / private
- Maps, slices (vectors), queues as first class members
- Pointers - without arithmetic
- No macros
- No exceptions
- No templates
- No operator overloading
- No warnings!

The syntax: Hello World

```
package main

import "fmt"

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

The syntax: Variables, looping

```
package main

import "fmt"

func main() {
    a, b := 1, 2

    fmt.Printf("a + b: %d\n", a + b)

     for a < 100 {
        a++
         fmt.Println(a)
     }
}
```

The syntax: Imports, casting

```
package main

import (
    "fmt"
    "math"
)

func main() {

     for i := 0; i < 100; i++ {
        fmt.Println(math.Pow(float64(i), 2))
     }
}
```

The syntax: Funcs

```
package main

import (
    "fmt"
//    "math"
)

func square(i int) int {
  return i*i
}

func main() {
    for i := 0; i < 100; i++ {
        fmt.Println(i, square(i))
    }
}
```

The syntax: Structs & Methods

```
type User struct {
        Name     string
        password string
}

// This is a method for User
func (u *User) Authenticate(name, password string) bool {
        return u.Name == name && u.password == password
}

// Instead of constructors...
func NewUser(name, password string) *User {
        return &User{  Name: name,  password: password,   }
}

// Instead of inheritence...
type Administrator struct {
        User
        email string
}

func (a *Administrator) SendMail(msg string) {
        log.Println("Sending message to ", a.Name)
}
```

Reason 2: Compiler Speed

- The whole SDK and compiler compiled in 19 seconds.
- Standard library - a couple of seconds. For 370 KLOC.
- My own project, 5 KLOC - 0.186 seconds.

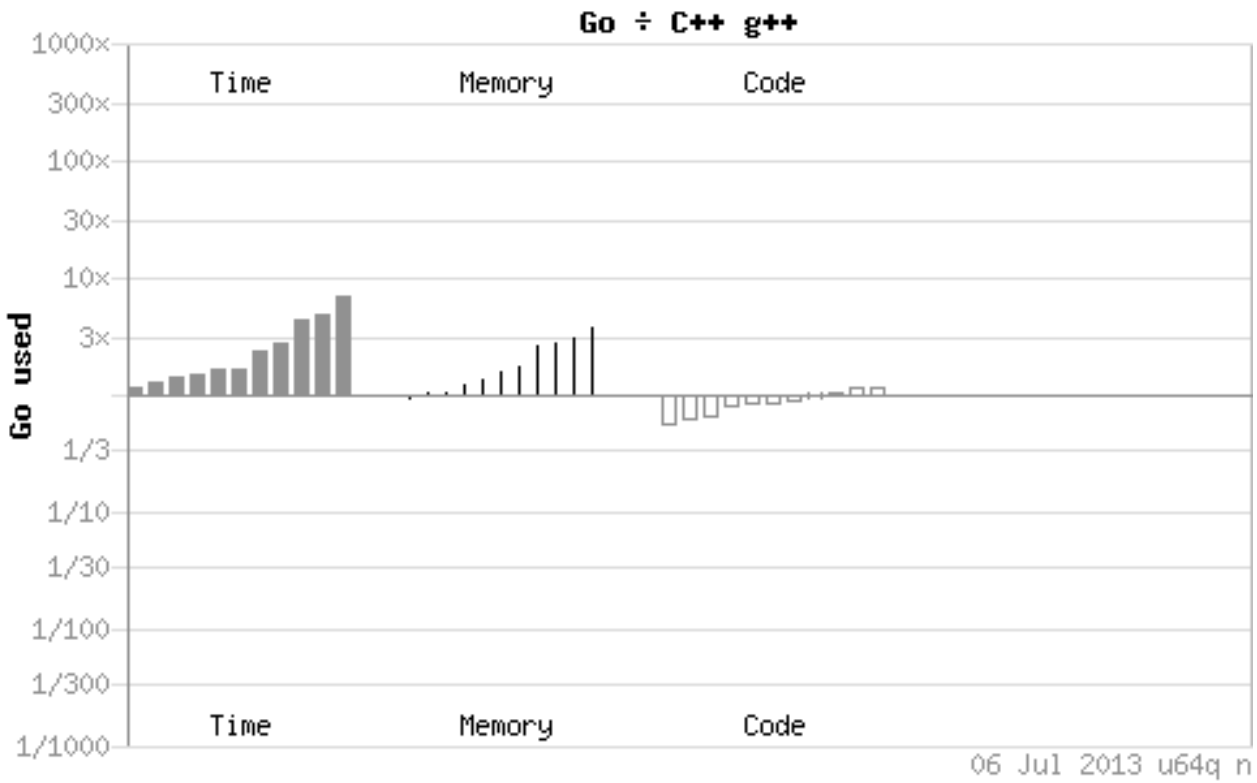
Why?

- Compiling imports only directly imported stuff.
- Compiler friendly syntax.
- The language is very small.
- Unused imports not allowed.
- Circular imports not allowed (no #IFDEF).

Reason 3: Execution Speed

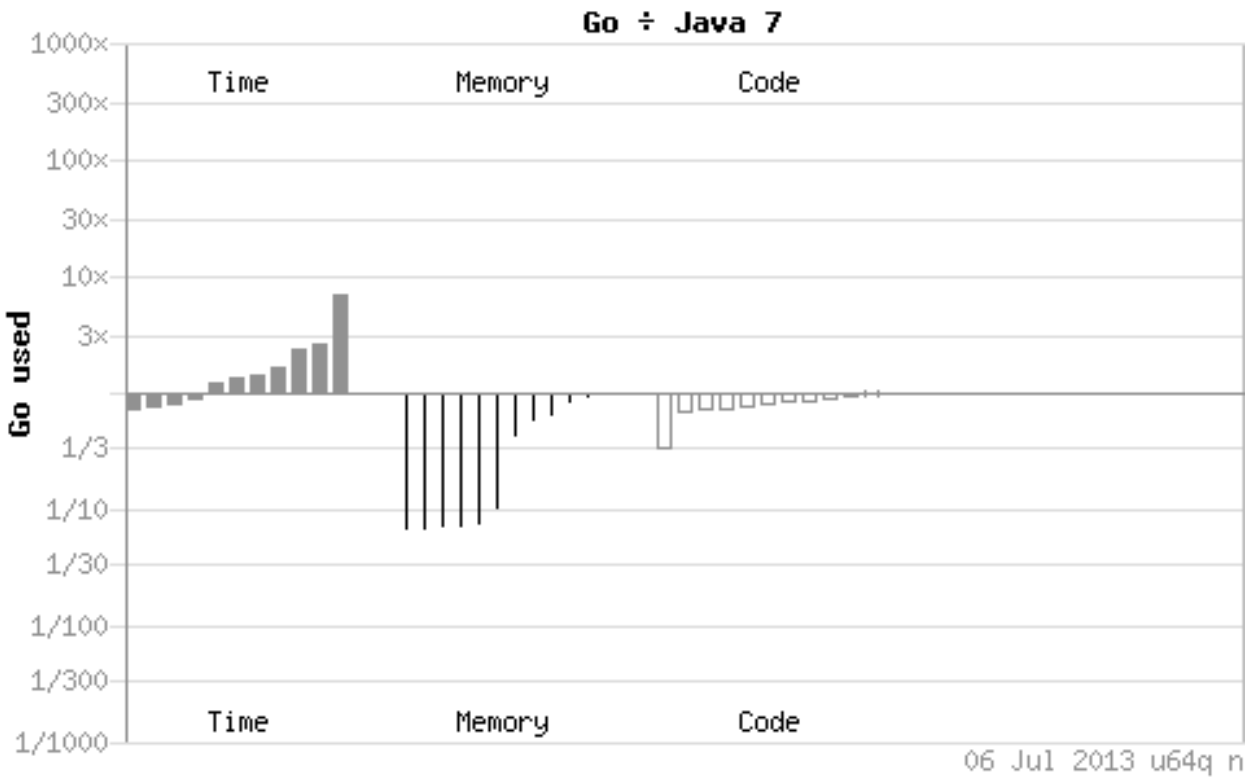
- Go is FAST
- Slower usually than C++, not by a lot.
- More or less on par with Java
- about x10 to x100 faster than Python/Ruby
- (but let's not forget concurrency)
- Only gcc-go does machine code optimizations
- Significant performance improvements between versions.

Execution Speed: Go vs. C++



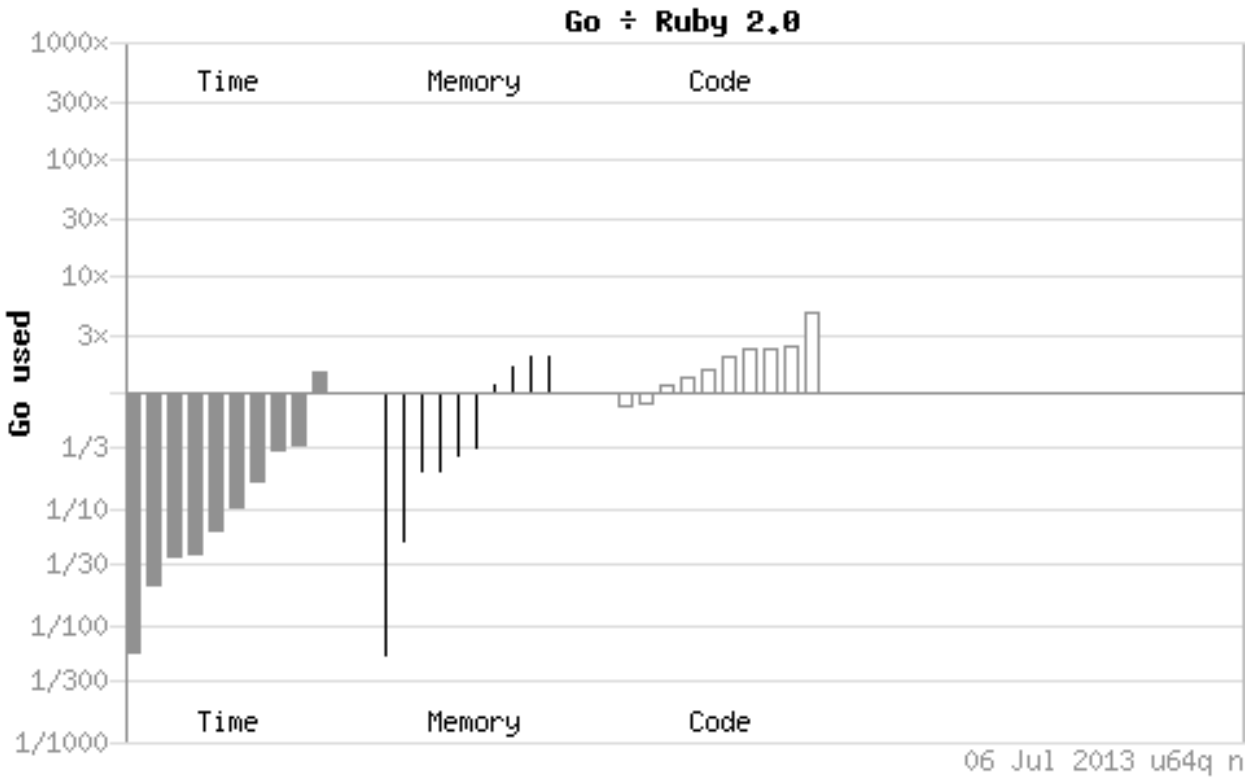
(Data: The benchmark game, http://benchmarksgame.alioth.debian.org/)

Execution Speed: Go vs. Java



(Data: The benchmark game, http://benchmarksgame.alioth.debian.org/)

Execution Speed: Go vs. Ruby



(Data: The benchmark game, http://benchmarksgame.alioth.debian.org/)

Reason 4: The Ecosystem

- Go Get - a pip/gem like installation system
- Compliant with Github / Google Code, Bitbucket
- Any repo can be made compliant
- The repo url is also the namespace!
- Can be automated to run on compile time
- Excellent repo search engines
- Friendly, useful community

Importing a package: Example

```
package main

import (
    "github.com/dvirsky/go-pylog/logging"
)

func main() {

    logging.Info("All Your Base Are Belong to %s!", "us")
    logging.Critical("And now with a stack trace")
}
```

Reason 5: Simple Conventions

- Lots of convention-over-configuration:
- all tests & benchmarks in *_test.go - just hit **go test**
- `func TestXXX(t *testing.T) {}`
- `func BenchmarkXXX(b *testing.B) {}`
- `func ExampleFoo() {}`

- Excellent documentation system built-in
- Folders are packages, URLs for installed
- Public / private

Documentation & Examples: Input

```
407
408 // MatchString checks whether a textual regular expression
409 // matches a string.  More complicated queries need
410 // to use Compile and the full Regexp interface.
411 func MatchString(pattern string, s string) (matched bool, err error) {
412         re, err := Compile(pattern)
413         if err != nil {
414                 return false, err
415         }
416         return re.MatchString(s), nil
417 }
418
```

```
23
24 func ExampleMatchString() {
25         matched, err := regexp.MatchString("foo.*", "seafood")
26         fmt.Println(matched, err)
27         matched, err = regexp.MatchString("bar.*", "seafood")
28         fmt.Println(matched, err)
29         matched, err = regexp.MatchString("a(b", "seafood")
30         fmt.Println(matched, err)
31         // Output:
32         // true <nil>
33         // false <nil>
34         // false error parsing regexp: missing closing ): `a(b`
35 }
36
```

Documentation & Examples: Output

func MatchString

```
func MatchString(pattern string, s string) (matched bool, err error)
```

MatchString checks whether a textual regular expression matches a string. More complicated queries need to use Compile and the full Regexp interface.

▾ Example

```
package main

import (
        "fmt"
        "regexp"
)

func main() {
        matched, err := regexp.MatchString("foo.*", "seafood")
        fmt.Println(matched, err)
        matched, err = regexp.MatchString("bar.*", "seafood")
        fmt.Println(matched, err)
        matched, err = regexp.MatchString("a(b", "seafood")
        fmt.Println(matched, err)
}
```

```
true <nil>
false <nil>
false error parsing regexp: missing closing ): `a(b`
```

Run | Format | Share

Reason 6: Concurrency

- Goroutines and Channels are the heart of Go
- Goroutines are microthreads with an internal scheduler
- You can run 10Ks of goroutines easily
- No need for non-blocking IO. It is under the hood!
- The usual pattern: One goroutine per server connection
- Locks are available but not encouraged

Concurrency: Channels

- Channels are synchronized message queues between goroutines.
- They are strictly typed first-class citizens.
- Delegate state through channels instead of sharing it.
- Buffered channels for non blocking pushes.
- They can be iterated on.

Channels: Over-simplified example...

```
func producer(c chan string) {
    for i := 0; i < 100; i++ {
        c <- fmt.Sprintf("Message #%d", i)
    }
    close(c)
}

func main() {
    c := make(chan string)
    go producer(c)

    for s := range c {
        fmt.Println(s)
    }
}
```

Channels: Slightly more complex #1

```
package main

import (
	"fmt"
	"runtime"
)

type result struct {
	in  int
	out int
}

//worker getting numbers and sending back their sqaure
func squarer(in chan int, out chan result) {
	for i := range in {
		out <- result{i, i * i}
	}
}

//take in the results and print them
func aggregator(res chan result) {
	for s := range res {
		fmt.Println(s)
	}
}
// TO BE CONTINUED ON NEXT SLIDE ------>
```

Channels: Slightly more complex #2

```
func main() {

	//utilize all CPUs
	runtime.GOMAXPROCS(runtime.NumCPU())

	//create the channels
	inChan := make(chan int)
	outChan := make(chan result)

	//fire up the workers
	for i := 0; i < 10; i++ {
		go squarer(inChan, outChan)
	}

	//aggregate results
	go aggregator(outChan)

	//push the input
	for i := 0; i < 1000; i++ {
		inChan <- i
	}

	//there are prettier ways of blocking here... :)
	var input string
	fmt.Scanln(&input)
}
```

Reason 7: The standard library

- Excellent, rich standard library
- More like Python's than C++'s
- Very well documented
- **Great** source for idiomatic code
- Excellent serialization

Reason 7: The standard library

- The "Batteries" include:
  - Robust HTTP Web Server + template library
  - Compression, encryption, JSON / XML / CSV
  - Profiling, debugging, source parsing
  - Reflection library
  - Image manipulation
  - Plus the usual suspects
  - No GUI Toolkit :)

The standard library: A Web Server

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

Reason 8: C is never too far

- CGo is part of the standard library
- Makes binding C libraries trivial
- You can also trivially call Go code from C
- BTW: A JVM based Go exists ;)

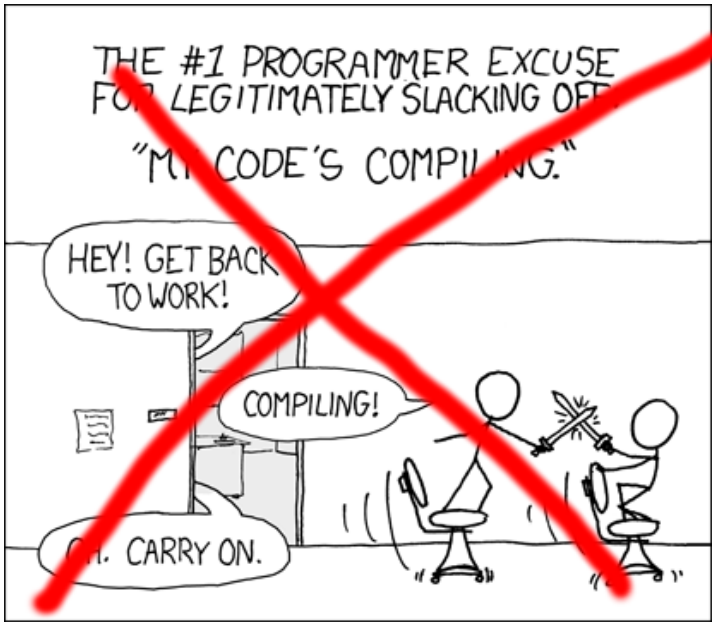
```
package rand

/*
#include <stdlib.h>
*/
import "C"

func Random() int {
    return int(C.rand())
}

func Seed(i int) {
    C.srand(C.uint(i))
}
```

Reason 9: Implicit Interfaces

- Implementing an interface simply means you implement it, no declaration needed.
- Example: any struct that has a method **String() string** can be used for "%s" fmt
- You can mix and match interfaces as you go
- The empty interface, **interface{}** is super useful.
- Interfaces can be safely cast into structs, or called directly.

Reason 10: ~~Procrastination~~ It's Fun!

- Go is fun to work on and read about
- Beware: productive procrastination :)
- "Code is compiling" is no longer valid!
- Thank God we have a new excuse



Before we get too excited

- Lack of complete, real IDE
- Debugging: GDB Works great - but no real frontend
- No dynamic linking/loading
- Implicit interfaces are cool but can be confusing
- No exceptions - a matter of taste.

Wanna write some Go code?

We're hiring! :)

We're introducing Go to our backend and data crunching stack, where C++ used to roam, or where Go could kick Python's ass.

Email jobs@everything.me

A Few Good Resources

- The official website:

  http://golang.org/

- Excellent introductory book:

  http://www.golang-book.com/

- GoDoc: Package search

  http://godoc.org/

- Migrating from Python to Go:

  http://blog.repustate.com/migrating-code-from-python-to-golang-what-you-need-to-know/2013/04/23/