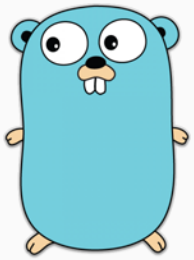
Programming in Go

"In Go, the code does exactly what it says on the page"

Matt Holiday

20 March 2019

Cardinal Peak



A better C from the folks who created Unix & C

“Go is more about software engineering than programming language research. Or to rephrase, it is about language design in the service of software engineering.” — Rob Pike

“Go is not meant to innovate programming theory. It’s meant to **innovate programming practice**.” — Samuel Tesla

“Go is like a better C, from the guys that didn’t bring you C++”
— Ikai Lan

Hello, world!

```
package main
import "fmt"

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

1, 2, 3, 5, 8, 13, 21, 34, 55, 89

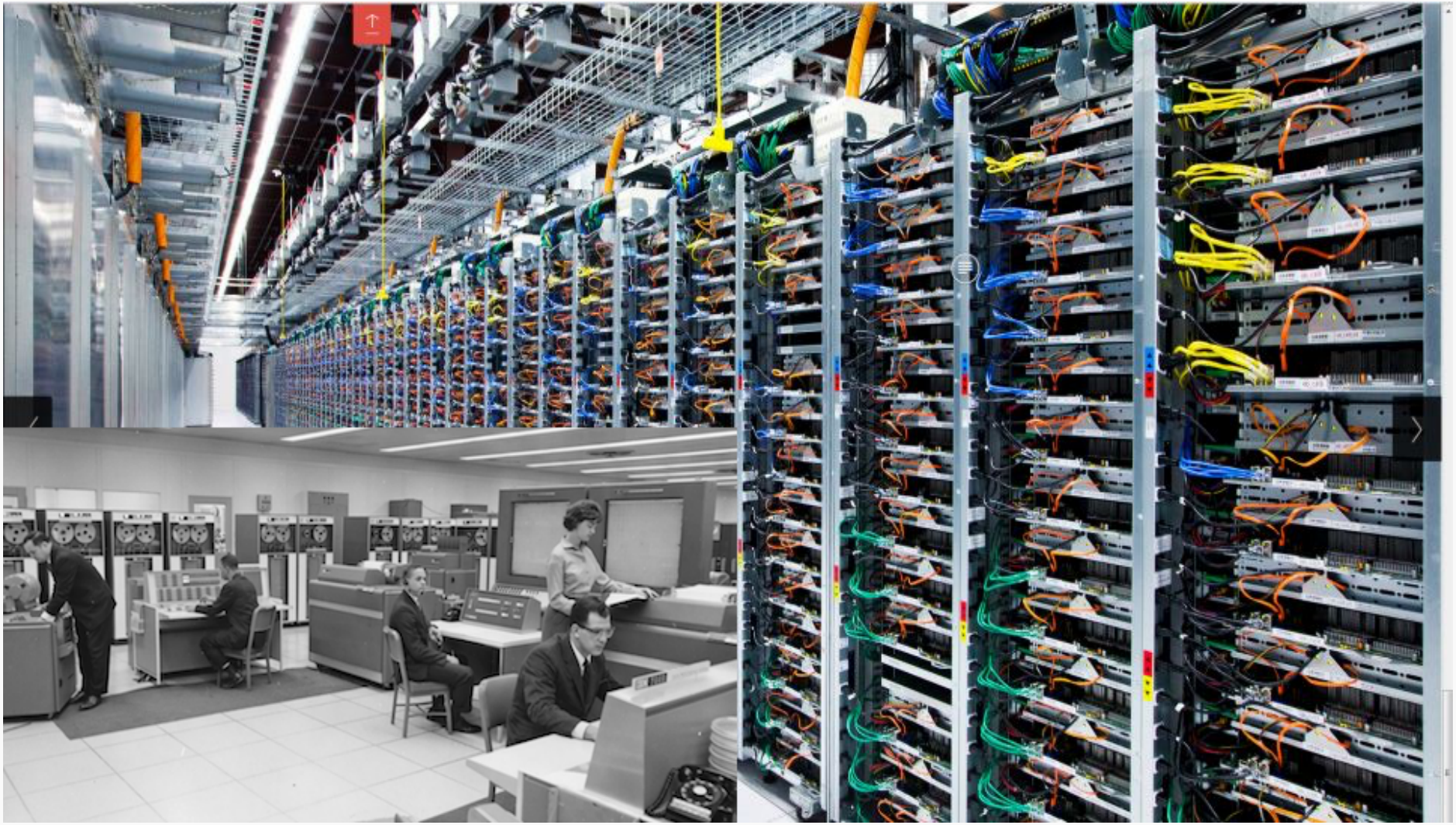
```
package main
import "fmt"

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

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

func main() {
    f := fib()
    for x := f(); x < 100; x = f() {
        fmt.Println(x)
    }
}
```

Why did they do it?



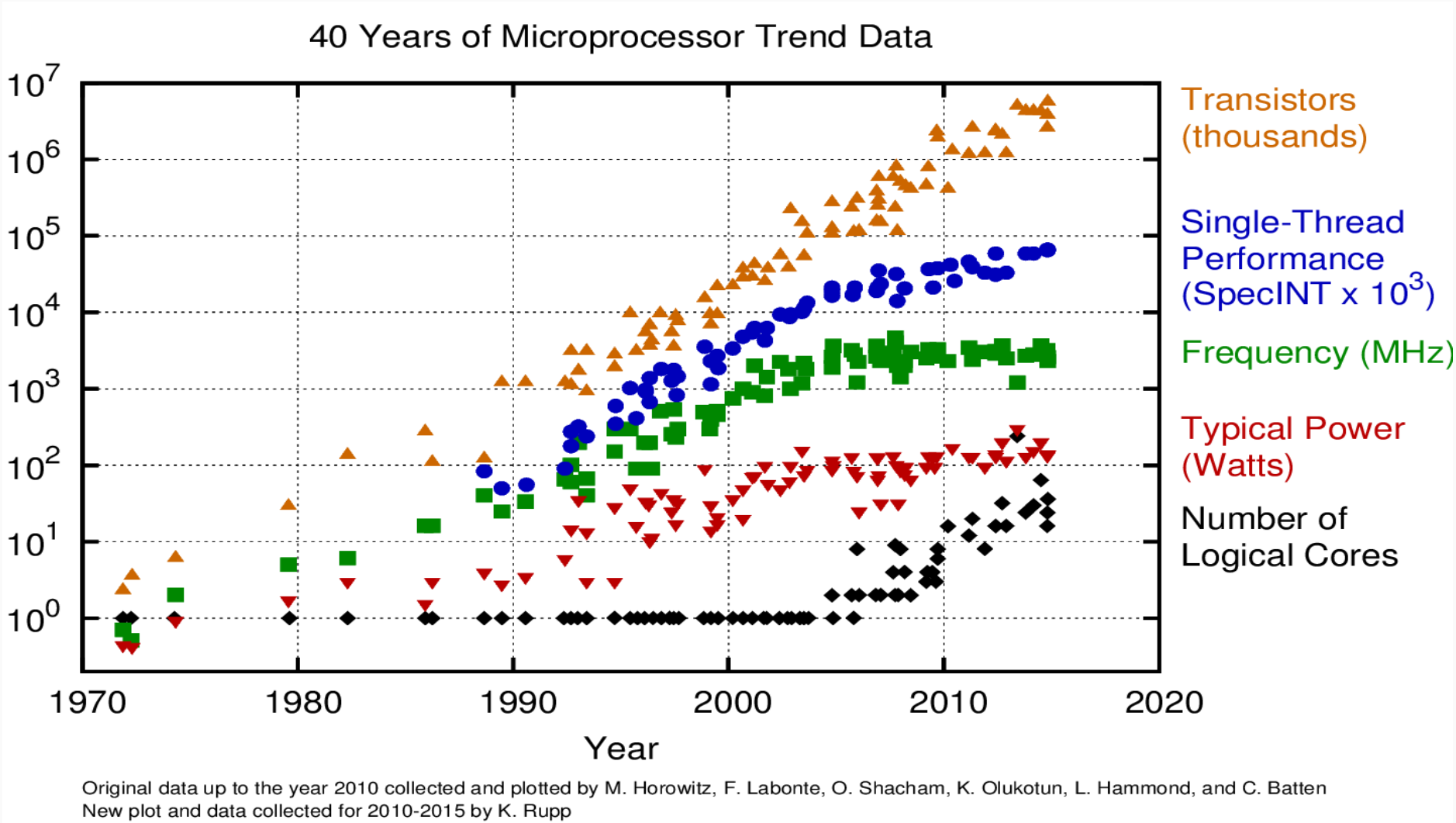
Why did they do it?

A new language for a new computing landscape:

- multicore processors
- networked systems
- massive clusters
- the web programming model (think REST)
- huge programs
- large numbers of developers
- long build times
- C++ and Java don't fit that landscape well

##	Language	Year
1	JavaScript	1995
2	Java	1995
3	Python	1991
4	C#	2000
5	C++	1983
6	C	1972

Why did they do it?



Why did they do it?

Pain points at Google:

- **slow builds**
- uncontrolled dependencies
- each programmer using a **different subset** of the language
- poor program understanding (code hard to read, poorly documented, and so on)
- duplication of effort
- cost of updates
- version skew
- **difficulty of writing static checkers & other tools**
- cross-language builds

Built for Google's problems: Distributed systems

It's taking over the infrastructure/container/cloud world:

- Docker
- Kubernetes
- Helm
- CoreOS (etcd, flannel)
- Prometheus
- Grafana
- CockroachDB
- DropBox
- CloudFlare

©Renée French

And I've delivered my first Go-based server to [redacted] ...

Design goals

Overall goals:

- **simplicity**, **safety**, and **readability**
- ease of expressing algorithms
- orthogonality
- one right way to do things

Necessary, simple features with predictable behavior:

- interfaces & methods
- packages
- concurrency
- tools & build speed

“Clear is better than clever” — Go Proverb

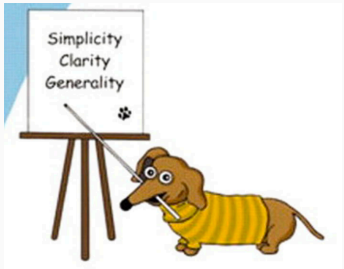
Simplicity

A language with fewer features $\Longrightarrow$ simplicity

Simplicity $\Longrightarrow$ readability $\Longrightarrow$ reliability & maintainability

Think of *The Practice of Programming:*

- Simplicity
- Clarity
- Generality



"If our basic tool, the language in which we design and code our programs, is also complicated, **the language itself becomes part of the problem rather than part of its solution.**" — Tony Hoare

Simplicity?

C	`#include <stdio.h>`	360 lines	9 files
C++	`#include <iostream>`	25326 lines	131 files
Go	`import "fmt"`	195 lines	1 file

Language	Year	Keywords	Pages
Pascal	1974	35	167
ANSI C	1988	32	238
C11	2011	44	683
C++	1990	48	480
C++14	2014	86	1330
Java 8	2015	50	788
Go	2012	25	98

C++20 may well be as big a release as C++11; draft over 1700 pp.

Odious comparisons

My subjective take:

- Go is as fast as Java and faster than interpreted languages
- with more safety but still very easy to use
- and offers radical simplicity

YMMV:

Language	Safety	Ease	Speed
Go	9	8	7
Java / C#	7	6	6
Python	5	9	1
Javascript	5	7	2
C++	3	5	8
C	1	3	9

“Fast languages aren’t safe, easy-to-use languages aren’t fast”

What It Is

Static but easy typing

“Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language.” — *Go FAQ*

“Clumsy type systems drive people to dynamically typed languages.” — Robert Griesemer

Look, Ma, no types!

```
package main
import ("io"; "log"; "os")

func main() {
    for _, fname := range os.Args[1:] {
        file, err := os.Open(fname)

        if err != nil {
            log.Fatal(err)
        }

        if _, err = io.Copy(os.Stdout, file); err != nil {
            log.Fatal(err)
        }

        file.Close()
    }
}
```

Concurrency based on Tony Hoare's CSP

Communicating sequential processes

Break your program into "nanoservices" that talk to each other

Like Erlang, but Go's channel model is more flexible and natural

"Go doesn't force developers to embrace the asynchronous ways of event-driven programming. ... That lets you **write asynchronous code in a synchronous style**. As people, we're much better suited to writing about things in a synchronous style." — Andrew Gerrand

Go's Concurrency Features

Goroutines — lightweight coroutines multiplexed onto threads

Channels (similar to shell pipes in Unix)

A `select` block allows multiplexing channels *in source code*

Think of channels as vehicles to transfer ownership

"Don't communicate by sharing memory; instead, **share memory by communicating.**" — Rob Pike

Concurrency Example 1: Channels

```
results := make(chan int)

// asynchronously yield a stream of integers

go func(limit int, out chan<- int) {
    for i := 0; i < limit; i++ {
        out <- i
    }

    close(out)  // else we would deadlock
}(10, results)

// receive them when they're ready

for i := range results {
    fmt.Println(i)
}
```

Concurrency Example 2: Stream of IDs

```
var nextID = 0

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v<h1>", nextID)

    // unsafe - data race

    nextID++
}

func main() {
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}

// simple HTTP server example from Francesc Campoy
```

Concurrency Example 2: Stream of IDs

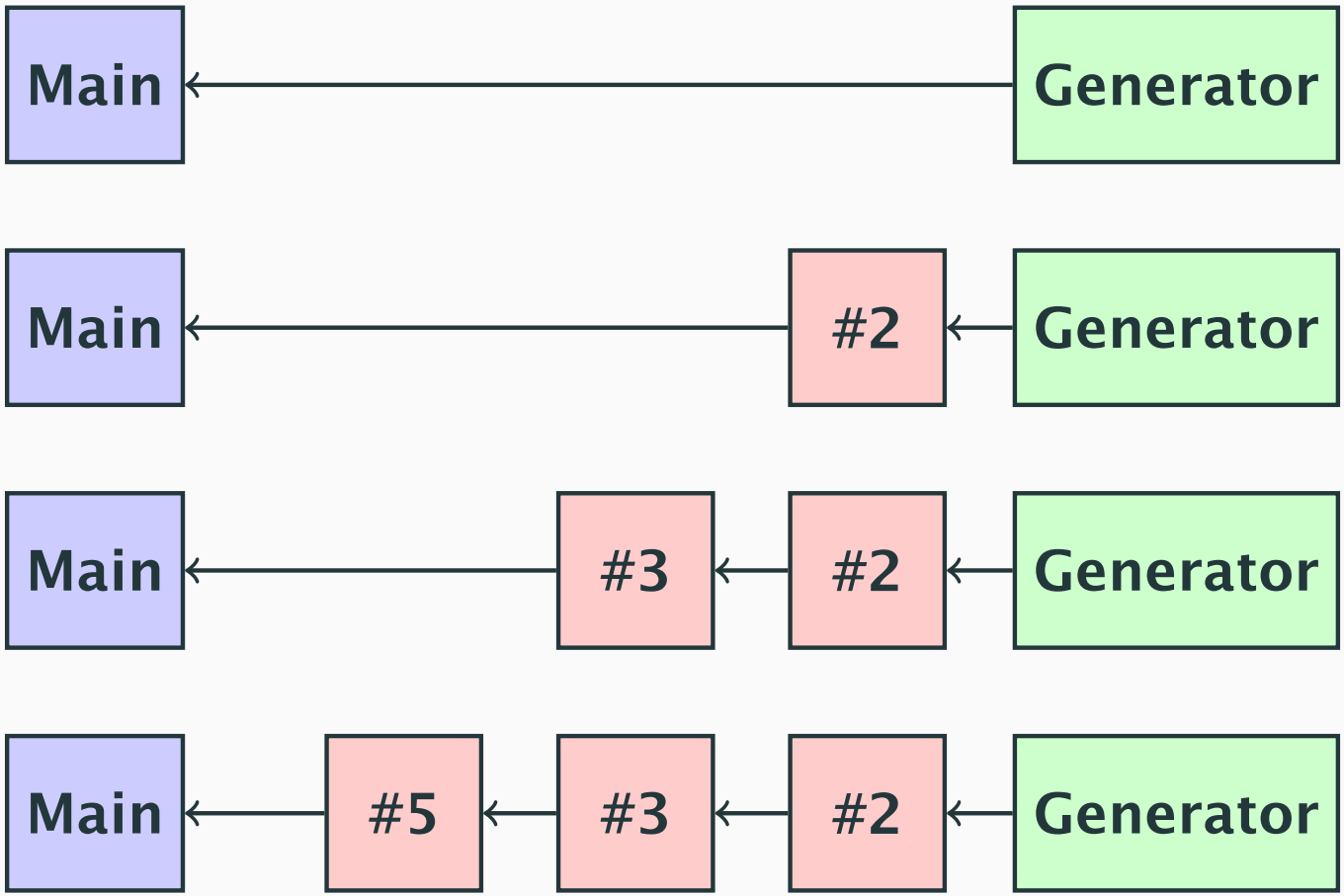
```
var nextID = make(chan int)

func handler(w http.ResponseWriter, q *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v<h1>", <-nextID)
}

func counter() {
    for i := 0; ; i++ {
        nextID <- i
    }
}

func main() {
    go counter()
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

Concurrency Example 3: Prime Sieve



Concurrency Example 3: Prime Sieve

```
// Doug McIlroy (1968) via Tony Hoare (1978)
// code example from the Go language spec

func generate(limit int, ch chan<- int) {
    for i := 2; i < limit ; i++ {
        ch <- i
    }
    close(ch)
}

func filter(src <-chan int, dst chan<- int, prime int) {
    for i := range src {
        if i % prime != 0 {
            dst <- i
        }
    }
    close(dst)
}
```

Concurrency Example 3: Prime Sieve

```
func sieve(limit int) {
    ch := make(chan int)
    go generate(limit, ch)
    for {
        prime, ok := <- ch
        if !ok {
            break
        }
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
        fmt.Print(prime, " ")
    }
}

func main() {
    sieve(100) // 2 3 5 7 11 13 17 19 23 29 31 37 41 43 ...
}
```

Object-orientation based loosely on Oberon-2

Type-bound functions (methods) *on any named type*

Abstraction using interfaces

Encapsulation using packages

Composition not inheritance

“Prefer composition to inheritance”
— Joshua Bloch, *Effective Java*

Object-Oriented Example 1: Interfaces

```
type Stringer interface { // pre-declared in Go
    String() string
}

type Pair struct {
    Path string
    Hash string
}

func (p Pair) String() string {
    return fmt.Sprintf("Hash of %v is %v", p.Path, p.Hash)
}

p := Pair{"/usr", "0xfdfe"}

fmt.Println(p) // calls p.String() automatically
               // "Hash of /usr is 0xfdfe"
```

Object-Oriented Example 1: Interfaces

```
// Sizer describes the Size() method that gets the
// total size of an item. [Mat Ryer]

type Sizer interface {
    Size() int64
}

func (f *File) Size() int64 {
    return f.info.Size()
}

func Fits(capacity int64, s Sizer) bool {
    return capacity > s.Size()
}

func IsEmailable(s Sizer) bool {
    return s.Size() < 1<<20
}
```

Object-Oriented Example 1: Interfaces

```
// add the ability to pass a slice of Sizers to Fits

type Sizers []Sizer

func (s Sizers) Size() int64 {
    var total int64

    for _, sizer := range s {
        total += sizer.Size()
    }

    return total
}

f1 := new(File) // returns *File
f2 := new(File)
ok := Fits(42, Sizers{f1, f2})
```

Object-Oriented Example 2: Composition

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

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

type ReadWriter interface {
    Reader
    Writer
}

// one of several ways to do this, all equal
type ReadWriteCloser interface {
    ReadWriter
    Close() error
}
```

Object-Oriented Example 2: Composition

```
func handler(w http.ResponseWriter, r *http.Request) {
    f, err := os.Open("./static/" + path.Base(r.URL.Path))

    if err != nil {
        http.Error(w, err.Error(), 404)
        return
    }


    // func Copy(dst Writer, src Reader) (int64, error)

    _, err := io.Copy(w, f)

    if err != nil {
        http.Error(w, err.Error(), 500)
    }
}
```

Fast compiles, small binaries, and lots of tools

Packages carry transitive import data $\Longrightarrow$ fast compiles

(**no** include [re]-pre-processing as in C)

Statically-linked binaries $\Longrightarrow$ tiny, safe containers

The language supports static analysis well:

- standard source formatting
- library includes all front-end parsing software

Built-in tools include:

- check style & correctness (golint, go vet)
- fetch 3rd-party packages (go get)
- generate documentation (go doc)

The compiler gives no warnings

Strictness $\Longrightarrow$ less ambiguity $\Longrightarrow$ fewer misteaks

- braces are required for **all** control statements
- no unused imports or variables
- no automatic integer/boolean conversions
- switch cases don't fall through by default
- assignment & increment are statements, not expressions

"**Form is liberating**" — quoted in *The Mythical Man-Month*

What else?

Garbage collection

Built-in race detector for testing

Unicode support (Go source is UTF-8)

Easy encode/decode of JSON, etc. using *reflection*

Tools and “unsafe” operations for interfacing to C (cgo)

A standard for code formatting supported by tools!
(~~brace wars~~)

What else?

No runtime dependencies:

- no interpreter or JVM
- if pure Go, no `libc` required
- no packages to install in container/VM

which reduces:

- stuff that needs to be managed
- start-up latency in lambdas
- security vulnerabilities

Go is **10–15 times faster** than interpreted languages (e.g., Python)

Reflection in action: JSON support

```
type Response struct {
    Page  int      `json:"page"`
    Words []string `json:"words,omitempty"`
}

r := &Response{Page: 1, Words: []string{"up", "lo", "an"}}

j, _ := json.Marshal(r)                    // ignoring errs
fmt.Println(string(j))

var r2 Response

json.Unmarshal(j, &r2)                     // ignoring errs
fmt.Printf("%#v\n", r2)

// {"page":1,"words":["up","lo","an"]}
// main.Response{Page:1, Words:[]string{"up", "lo", "an"}}
```

Reflection in action: JSON support

```
type Response struct {
    Page  int      `json:"page"`
    Words []string `json:"words,omitempty"`
}

r := &Response{Page: 1}

j, _ := json.Marshal(r)                    // ignoring errs
fmt.Println(string(j))

var r2 Response

json.Unmarshal(j, &r2)                     // ignoring errs
fmt.Printf("%#v\n", r2)

// {"page":1}
// main.Response{Page:1, Words:[]string{nil}}
```

What It Isn’t

What we might miss

No support for generics (coming in 2.0?)

Not much of a container class library

No constructors, destructors, etc.

No structured “const” objects (only numbers, strings)

No operator/function overloading, optional parameters;
many other **convenient/confusing** things were left out, e.g. `?:`

"Go’s design aims for being easy to use, which means it must be **easy to understand**, even if that sometimes contradicts superficial ease of use." — Rob Pike

What we won't miss

Circular module dependencies

Lasagna code

Security vulnerabilities from pointers

Language lawyers

Hundreds of MB of virtual machine code

Macro processing hell

Expensive but limited 3rd-party static checkers

```
/* This works because "Hello"[5] == 5["Hello"] */
```

Things that go bump in the night

Structural typing of interfaces
(who implements it? who *accidentally* implements it?)

Gotcha when making a slice from an underlying array
(length vs capacity; unintuitive behavior when appending)

Nil interface versus nil pointer in an interface (!)

Accidentally shadowing a variable with the `:=` operator

Append to a nil slice works, but not insert into a nil map

Interesting new classes of concurrency bugs

Some Remarks

Two views of Go: #1

"The rub is that it ignores the last 15–20 year of compiler/language research. To me, it's effectively an assembly language with CSP and implicit duck-typing. ... [It's] opaque to formal reasoning and as result merely promotes more ad-hoc programs and systems."
— Michael Lee

General complaints:

- too opinionated
- stuck in the 1970s
- **stuck in Unix thinking** 
- too simple / not enough syntactic sugar
- doesn't have [my favorite language feature]
- not a functional language / not a "true" OO language

Two views of Go: #2

"We use Go because it's boring. ... Go makes it easy to write code that is understandable. There's no 'magic' ... and none of the cute tricks you'll find in most Python or Ruby codebases. The code is **verbose but readable, unsophisticated but intelligible, tedious but predictable**." — Tyler Treat

"Go [might] sound rather dull and industrial, but in fact the focus on **clarity, simplicity and composability** throughout the design instead resulted in a **productive, fun language** that many programmers find expressive and powerful." — Rob Pike

Testimonials

"**Go makes much more sense** for the class of problems that C++ was originally intended to solve." — Bruce Eckel *(author & founding member, C++ standard committee)*

"If you're building a server, I can't imagine using anything other than Go. ... Node is not the best system to build a massive web server. **I would use Go for that**. And honestly, that's the reason why I left Node." — Ryan Dahl *(creator of Node.js)*

"Go is designed for the C-like jobs Python can't handle. ... If I were clean-starting an NTP implementation today, **I'd do it in Go** without any hesitation at all." — Eric S. Raymond *(author & advocate)*

OK, one more view

"A language that doesn't have everything is actually easier to program in than some that do." — Dennis Ritchie

Would you like a short course on Go?