Another Go at Language Design

Jazoon 2015

Marcel van Lohuizen
Google Switzerland GmbH

Another Go at Language Design

Marcel van Lohuizen
Google

Introducing Go:

```
package main

import "fmt"

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

History

Design began in late 2007

Initially:
- Robert Griesemer, Rob Pike, Ken Thompson

Soon thereafter:
- Later: Ian Lance Taylor, Russ Cox

Core team is currently about 25 people. Many other contributors inside and outside of Google.

Became open source in November 2009.

Developed entirely in the open; very active community.

Language stable as of Go 1, early 2012.

Who am I

Marcel van Lohuizen

Joined Google in 2002:

- search engine, infrastructure, Borg cluster management, etc. etc.
- borgcfg

Joined the Go team April 2011:

- main focus: Unicode handling, text processing, rethinking i18n and l10n.

Before Go

Compiled, statically-typed languages (C, C++, Java): effective, but clumsy:

- verbose, lots of repetition (too much typing)
- types get in the way as much as they help (too much typing)
- compiles take far too long

Dynamic languages (Python, JavaScript) fix some issues but introduce others:

- runtime errors that should be caught statically
- no compilation means slow code
- break down at scale

Neither are well-suited for modern architectures (all mostly >10 years old!)

There is a reason we see so many new languages!

About Go

Go has the lighter feel of a dynamically typed language but is compiled

- general purpose
- concise syntax
- expressive type system
- concurrency
- garbage collection
- fast compilation
- efficient execution

Go's purpose is *not* research into programming language design.

More about consolidating and reflecting upon existing features.

Make programming fun again!

Software developement at Google

Requirements

Must work at scale:

- large programs
- large teams
- large number of dependencies

Modernize:

- suitable for multicore machines
- suitable for networked machines
- suitable for web stuff

Topics

The things Go improves upon, significantly:

1. Dependencies
2. Readability
3. Object-Oriented Programming
4. Concurrency
5. Tooling

Will show a little bit of Go code along the way.

Topic 1: Dependencies

Build speed and dependencies

- Hours to build a binary on a single machine.

```
Example of some Google binary:
- 2000 files
- 4.2 megabytes
- 8 gigabytes delivered to compiler
- 2000 bytes sent to compiler for every C++ source byte
- it's real work too: `<string>` for example
```

- A complex distributed build system takes this down to minutes.
- But still no fun.

New clean compiler worth ~5X compared to gcc, but we need much more.

Dependencies in Go

Dependencies are defined (syntactically) in the language.

Efficient:

- dependencies traversed once per source file.
- package's object file contains all information needed to import it
- no recursive imports/includes
- export data early in object file

Result: exponentially less data read than with `#include` files:

- with Go in Google, about 40X fanout vs 2000x for C++

Plus, in C++ it's general code that must be parsed, in Go just export data.

Packages

Similar to Modula-2/ Oberon 2.

Combines properties of library, name space, and module.

By convention, a package maps to a directory with one or more source files

Unused dependencies and circular imports cause an error at compile time.

A Hello Foo web server

```
package main

import (
    "fmt"
    "net/http"
)

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

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

Run

localhost:8080/Jazoon (http://localhost:8080/Jazoon)

Remote packages

Package path syntax works with remote repositories.
The import path is just a string.

Can be a file, can be a URL:

```
go get github.com/coreos/go-etcd/etcd   // Command to fetch package

import "github.com/coreos/go-etcd/etcd" // etcd client's import statement

var client etcd.Conn         // Client's use of package
```

Visibility rules

Simple:

- upper case initial letter: `Name` is visible to clients of package
- otherwise: `name` (or `_Name`) is not visible to clients of package

Applies to variables, types, functions, methods, constants, fields....

That Is It.

Not an easy decision.

One of the most important things about the language.

Can see the visibility of an identifier without discovering the declaration.

Clarity.

Locality scales

Names do not leak across boundaries

- adding an exported name to my package cannot break your package!

No surprises when importing

In C, C++, Java the name y could refer to anything
In Go, y (or even Y) is always defined within the package.
In Go, x . Y is clear: find x locally, Y belongs to it.

Immediate consequences for readability.

Topic 2: Readability

The readability of programs is immeasurably more important than their writeability.
Hints on Programming Language Design
C. A. R. Hoare 1973

Syntax is not important... - unless you are a programmer or writing tools
Rob Pike.

Tenets of Go’s design

Simplicity

```
Each language feature should be easy to understand.
```

Orthogonality

```
Go’s features should interact in predictable and consistent ways.
```

Readability

```
What is written on the page should be comprehensible with little context.
```

Go's core libraries are designed with the same principles.

Too verbose

```
scoped_ptr<logStats::LogStats>
    logStats(logStats::LogStats::NewLogStats(FLAGS_logStats, logStats::LogStats::kFIFO));
```

-- observed code

```
public static <I, O> ListenableFuture<O> chain(ListenableFuture<I> input,
Function<? super I, ? extends ListenableFuture<? extends O>> function)
dear god make it stop
```

-- an observed log chat

Too dense

```
(n: Int) => (2 to n) |> (r => r.foldLeft(r.toSet)((ps, x) =>
    if (ps(x)) ps -- (x * x to n by x) else ps))
```

Just right

```
t := time.Now()
switch {
case t.Hour() < 12:
    return "morning"
case t.Hour() < 18:
    return "afternoon"
default:
    return "evening"
}
```

Syntax

Tooling is essential, so Go has a clean syntax.
Not super small, just clean:

- regular (mostly)
- only 25 keywords
- straightforward to parse (no type-specific context required)
- easy to predict, reason about

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Declarations

Uses Pascal/Modula-style syntax: name before type, more type keywords.

```
var fn func([]int) int
type T struct { a, b int }
```

not

```
int (*fn)(int[]);
struct T { int a, b; }
```

Easier to parse—no symbol table needed.

One nice effect: can drop `var` and derive type of variable from expression:

```
var buf *bytes.Buffer = bytes.NewBuffer(x) // explicit
buf := bytes.NewBuffer(x)                  // derived
```

For more information:

golang.org/s/decl-syntax (http://golang.org/s/decl-syntax)

Function syntax

Function on type T:

```
func Abs(t T) float64
```

Method of type T:

```
type T float64

func (t T) Abs() float64
```

Variable (closure) of type T:

```
negAbs := func(t T) float64 { return -Abs(t) }
```

In Go, functions can return multiple values. Common case: `error`.

```
func ReadByte() (c byte, err error)

c, err := ReadByte()
if err != nil { ... }
```

Error handling

Go has no `try-catch` control structures for exceptions.

Return `error` instead: built-in interface type that can "stringify" itself:

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

Clear and simple.

Philosophy:

Forces you think about errors—and deal with them—when they arise.
Errors are *normal*. Errors are *not exceptional*.

Use the existing language to compute based on them.
Don't need a sublanguage that treats them as exceptional.

Result is better code (if more verbose).

Topic 3: Object-oriented programming

What is object-oriented programming?

"Object-oriented programming (OOP) is a programming paradigm using *objects* – usually instances of a class – consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance. Many modern programming languages now support forms of OOP, at least as an option."

(Wikipedia)

Object orientation

Go is object-oriented.

Go does not have classes or subtype inheritance.

What does this mean?

O-O and program evolution

O-O is important because it provides uniformity of interface.

Problem: subtype inheritance encourages non-uniform interfaces.

Design by type inheritance oversold.

- generates brittle code.
- encourages overdesign early on
- there might be no natural hierarchy
- early decisions hard to change, often poorly informed.
- makes every programmer an interface designer.

Complicates designs.

Interfaces

Like interfaces in Java, but very different

In Go an interface is *just* a set of methods:

```
type Abser interface {
    Abs() float64
}
```

Implementations satisfy an interface implicitly. (Statically checked.)

The following type implements this interface:

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return -f
    }
    return f
}
```

There is no explicit hierarchy and thus no need to design one!

Interfaces in practice: composition

Tend to be small: one or two methods are common.

Composition naturally follows from small interfaces. Example, from package `io`:

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

`Reader` make it easy to chain:

- files, buffers, networks, encryptors, compressors, GIF, JPEG, PNG, ...

A type can implement many interfaces.

Interfaces are often introduced after the fact.

Composing with functions instead of methods

Hard to overstate the effect that Go's interfaces have on program design.

One big effect: functions with interface arguments.

```
func ReadAll(r io.Reader) ([]byte, error)
```

Wrappers or Decorators:

```
func LoggingReader(r io.Reader) io.Reader
func LimitingReader(r io.Reader, n int64) io.Reader
func ErrorInjector(r io.Reader) io.Reader
```

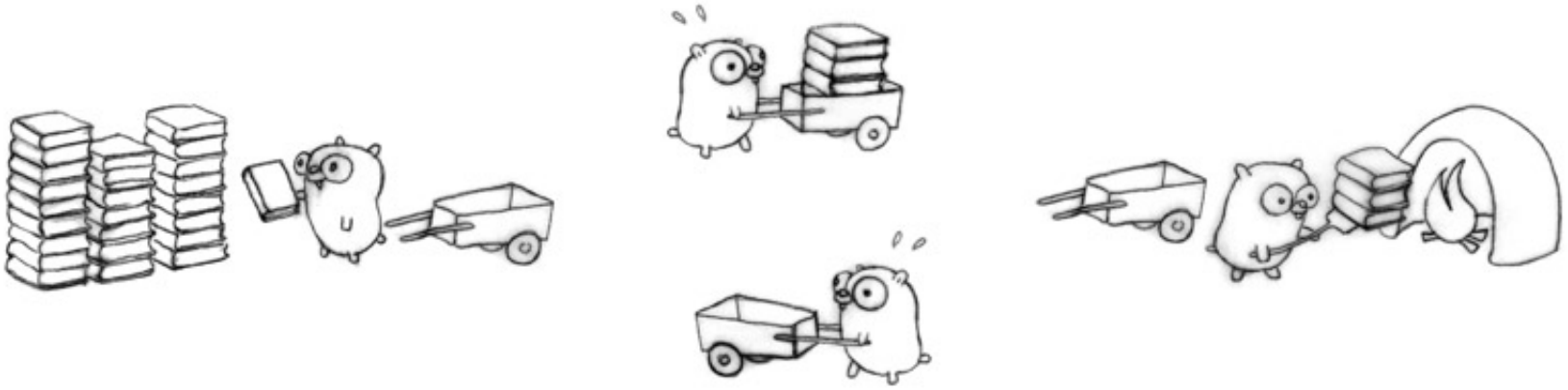
The designs are nothing like hierarchical, subtype-inherited methods.
Much looser, organic, decoupled, independent.

Chaining io.Readers

```
package main

import (
    "compress/gzip"
    "encoding/base64"
    "io"
    "os"
    "strings"
)

func main() {
    var r io.Reader
    r = strings.NewReader(data)
    r = base64.NewDecoder(base64.StdEncoding, r)
    r, _ = gzip.NewReader(r)
    io.Copy(os.Stdout, r)
}

const data = `
H4sIAAAJbogA/1SOO5KDQAxE8zlFZ5tQXGCjjfYIjoURoPKgcY0E57f4VZlQXf2e+r8yOYbMZJhoZWRxz3wkCVjeReETS0VH
z5fBCzpxxg/PbfrT/gacCjbjeiRNOChaVkA9RAdR8eVEw4vxa0Dcs3Fe2ZqowpeqG79L995l3VaMBUV/02OS+B6kMWikwG51
c8n5GnEPr11F2/QJAAD//z9IppsHAQAA
`
```

Run

Topic 4: Concurrency



Server software: threads vs event-driven callbacks

Thread per connection

- nice control flow
- doesn't scale in practice

Event-driven callbacks

- fast
- unclear flow: callback hell

Go's concurrency features are like using threads, but scalable:

```
for {
    conn, err := listener.Accept()
    // handle err
    go serve(conn)
}
```

Communicating Sequential Processes

Go is concurrent, in the CSP family.

Why CSP?

- the rest of the language can be ordinary and familiar.
- great fit for a web server, the canonical Go program.

Go provides independently executing **goroutines** that communicate and synchronize using **channels**.

Analogy with Unix: processes connected by pipes.

But in Go things are fully typed and lighter weight.

Goroutines

Start a new, concurrent flow of control with the go keyword:

```
func main() {
    go expensiveComputation(x, y, z)
    anotherExpensiveComputation(a, b, c)
}
```

"go" is like a '&' in Unix shell for funcs

A **goroutine** is like a thread, but lighter weight:

- goroutines are multiplexed onto OS threads
- stacks are small and sized on demand (key to scalability!!!)
- requires support from language, compiler, runtime
- can't just be a library

Channels

Our trivial parallel program again:

```
func main() {
    go expensiveComputation(x, y, z)
    anotherExpensiveComputation(a, b, c)
}
```

Need to know when the computations are done.

Need to know the result.

A Go **channel** provides the capability: a typed synchronous communications mechanism.

Channels

Goroutines communicate using channels.

```
func work(ch chan int, x int) {
    ch <- x * x
}

func main() {
    ch := make(chan int)
    go work(ch, 2)
    go work(ch, 5)
    fmt.Println(<-ch, <-ch)
}
```

Run

Share memory by communicating

Traditionally, you implement a worker pool by sharing memory and synchronizing with mutexes.

In Go, you reverse the equation.

```
type Work struct { x, y, result int }

func worker(in <-chan *Work, out chan<- *Work) {
    for w := range in {
        w.result = w.x * w.y // do some work...
        out <- w
    }
}
func main() {
    in, out := make(chan *Work), make(chan *Work)
    for i := 0; i < 10; i++ { go worker(in, out) }
    go sendLotsOfWork(in)
    receiveLotsOfResults(out)
}
```

Performance and scaling demo

How fast is it to start and stop 100K concurrent goroutines?

```
func main() {
    ch := make(chan int)       // Typical use of channel.
    barrier := make(chan bool) // Using channel as barrier.

    f := func() {
        <-barrier // Block until barrier is "released".
        ch <- 1   // Send a value.
    }

    const n = 100000 // launch a hundred thousand goroutines
    start := time.Now()
    for i := 0; i < n; i++ {
        go f()
    }
    close(barrier) // signal them to start sending

    for i := 0; i < n; i++ {
        <-ch
    }
    fmt.Printf("%v to launch %d concurrent goroutines", time.Since(start), n)
}
```

Run

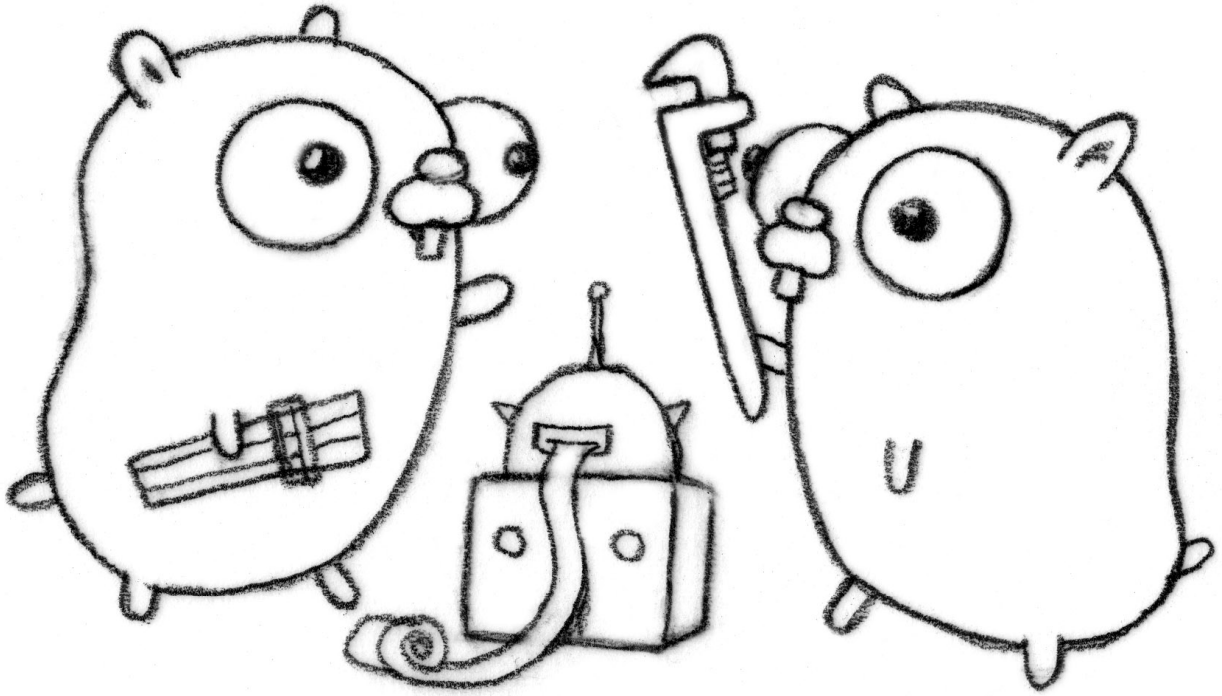
Select

A select statement is like a switch, but it selects over channel operations (and chooses exactly one of them).

```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.NewTicker(250 * time.Millisecond)
    boom := time.After(1 * time.Second)
    for {
        select {
        case <-tick.C:
            fmt.Println("tick")
        case <-boom:
            fmt.Println("BOOM!!!")
            return
        }
    }
}
```

Run

Topic 5: Tooling



Tools

Software engineering requires tools.

Go's syntax, package design, naming, etc. make tools easy to write.

Standard library includes lexer, parser, and type checker.

Oracle

Gofmt

Always intended to do automatic code formatting.
Eliminates an entire class of argument.
Runs as a "presubmit" to the code repositories.

Training:

- The community has always seen `gofmt` output.

Sharing:

- Uniformity of presentation simplifies sharing.

Scaling:

- Less time spent on formatting, more on content.

Often cited as one of Go's best features.

Also

Testing

- Go cover
- Race detecter
- Go fuzz

And more

- profiling
- goroutine analyzer
- etc.

Other reasons for using Go

Other reasons for using Go:

Easy deployment

- static binary
- cross compilation

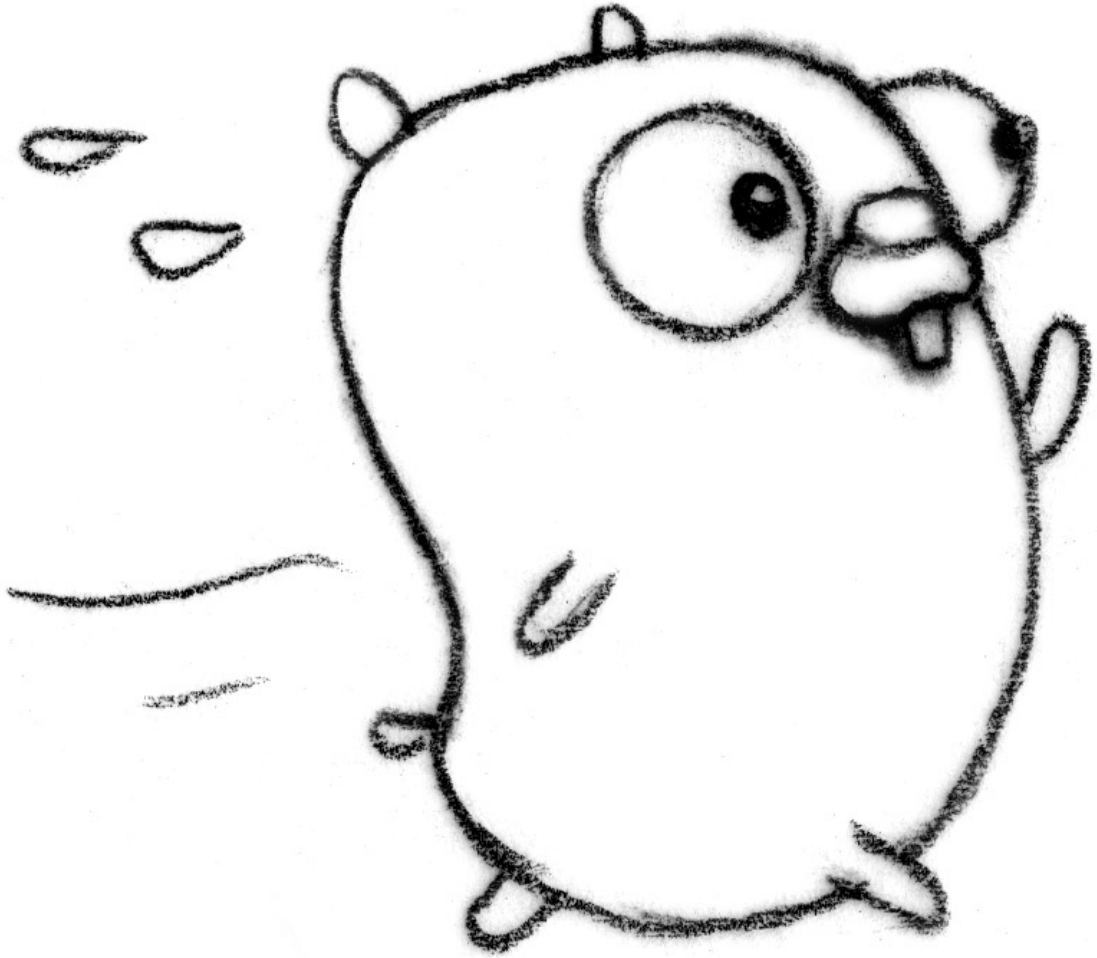
High-quality core library:

- net, http, crypto, images, and much more

Renaissance of command line tools: fast start times

Getting popular as a scripting language

Cute, but is Go actually running anywhere?



At Google

- Kubernetes
- dl.google.com
- Youtube (MySQL scaling infrastructure)
- Flywheel (data compression)

many others

At Scale: Baidu

> 100B requests/day

~ 1.15M qps

100% Go HTTP server

At Scale: Dropbox

Betting the farm

- Bigcommerce
- Chain.com
- CloudFlare
- CoreOS
- Docker
- Drone.io
- Ethereum
- HailO
- Iron.io
- Hashicorp (switching from Ruby)
- Juju
- SendGrid

Many others

- Apple
- Basecamp
- Canonical
- Facebook
- Heroku
- Pivotal Labs
- Square

Switch Success Stories

Parse

```
"How We Moved Our API From Ruby to Go and Saved Our Sanity"

"Was the rewrite worth it? Hell yes it was. Our reliability improved by an order of magnitude."
```

Repustate

```
"From Python to Go: migrating our entire API"

"reducing the mean response time of an API call from 100ms to 10ms"
```

Google

```
dl.google.com
```

Conclusion

Safe, fast, and scalable like statically typed languages, but fast development cycle like dynamically typed languages.

Simple, clean concurrency model, with the performance of event-driven callbacks.

Better tooling: more automation

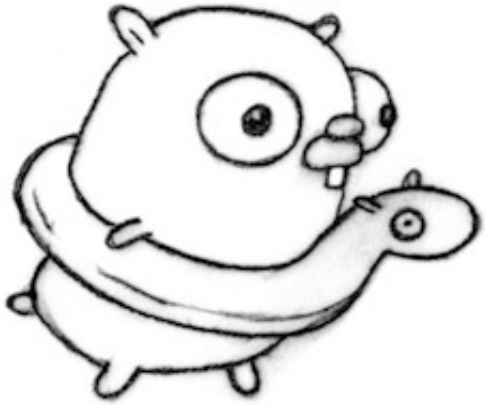
Simplified but powerful abstractions

Adoption growing quickly

Try it!

Productive in one day,
Efficient in one week,
Expert in one year.
-- Andrew Gerand

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



Thank you

Marcel van Lohuizen
Google Switzerland GmbH

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

@mpvanlohuizen (http://twitter.com/mpvanlohuizen)