

Go for gophers

GopherCon closing keynote
25 April 2014

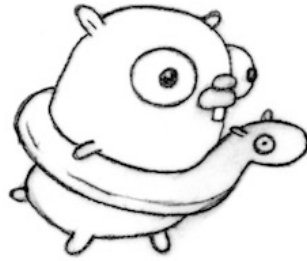
Andrew Gerrand
Google, Inc.

Video

A video of this talk was recorded at GopherCon in Denver.

[Watch the talk on YouTube](https://www.youtube.com/watch?v=dKGmK_Z1Zl0) (https://www.youtube.com/watch?v=dKGmK_Z1Zl0)

About me



I joined Google and the Go team in February 2010.

Had to re-think some of my preconceptions about programming.

Let me share what I have learned since.

Interfaces

Interfaces: first impressions

I used to think about classes and types.

Go resists this:

- No inheritance.
- No subtype polymorphism.
- No generics.

It instead emphasizes *interfaces*.

Interfaces: the Go way

Go interfaces are small.

```
type Stringer interface {  
    String() string  
}
```

A `Stringer` can pretty print itself.

Anything that implements `String` is a `Stringer`.

An interface example

An `io.Reader` value emits a stream of binary data.

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

Like a UNIX pipe.

Implementing interfaces

```
// ByteReader implements an io.Reader that emits a stream of its byte value.  
type ByteReader byte  
  
func (b ByteReader) Read(buf []byte) (int, error) {  
    for i := range buf {  
        buf[i] = byte(b)  
    }  
    return len(buf), nil  
}
```

Wrapping interfaces

```
type LogReader struct {
    io.Reader
}

func (r LogReader) Read(b []byte) (int, error) {
    n, err := r.Reader.Read(b)
    log.Printf("read %d bytes, error: %v", n, err)
    return n, err
}
```

Wrapping a ByteReader with a LogReader:

```
r := LogReader{ByteReader('A')}
b := make([]byte, 10)
r.Read(b)
fmt.Printf("b: %q", b)
```

[Run](#)

By wrapping we compose interface *values*.

Chaining interfaces

Wrapping wrappers to build chains:

```
var r io.Reader = ByteReader('A')
r = io.LimitReader(r, 1e6)
r = LogReader{r}
io.Copy(ioutil.Discard, r)
```

More succinctly:

```
io.Copy(ioutil.Discard, LogReader{io.LimitReader(ByteReader('A'), 1e6)})
```

[Run](#)

Implement complex behavior by composing small pieces.

Programming with interfaces

Interfaces separate data from behavior.

With interfaces, functions can operate on *behavior*:

```
// Copy copies from src to dst until either EOF is reached
// on src or an error occurs. It returns the number of bytes
// copied and the first error encountered while copying, if any.
func Copy(dst Writer, src Reader) (written int64, err error) {
```

```
io.Copy(ioutil.Discard, LogReader{io.LimitReader(ByteReader('A'), 1e6)})
```

[Run](#)

Copy can't know about the underlying data structures.

A larger interface

`sort.Interface` describes the operations required to sort a collection:

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

`IntSlice` can sort a slice of ints:

```
type IntSlice []int

func (p IntSlice) Len() int           { return len(p) }
func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p IntSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
```

`sort.Sort` uses can sort a `[]int` with `IntSlice`:

```
s := []int{7, 5, 3, 11, 2}
sort.Sort(IntSlice(s))
fmt.Println(s)
```

[Run](#)

Another interface example

The `Organ` type describes a body part and can print itself:

```
type Organ struct {
    Name    string
    Weight  Grams
}


func (o *Organ) String() string { return fmt.Sprintf("%v (%v)", o.Name, o.Weight) }

type Grams int

func (g Grams) String() string { return fmt.Sprintf("%dg", int(g)) }

func main() {
    s := []*Organ{{"brain", 1340}, {"heart", 290},
                  {"liver", 1494}, {"pancreas", 131}, {"spleen", 162}}

    for _, o := range s {
        fmt.Println(o)
    }
}
```



Sorting organs

The `Organs` type knows how to describe and mutate a slice of organs:

```
type Organs []*Organ

func (s Organs) Len() int { return len(s) }
func (s Organs) Swap(i, j int) { s[i], s[j] = s[j], s[i] }
```

The `ByName` and `ByWeight` types embed `Organs` to sort by different fields:

```
type ByName struct{ Organs }

func (s ByName) Less(i, j int) bool { return s.Organs[i].Name < s.Organs[j].Name }

type ByWeight struct{ Organs }

func (s ByWeight) Less(i, j int) bool { return s.Organs[i].Weight < s.Organs[j].Weight }
```

With embedding we compose *types*.

Sorting organs (continued)

To sort a `[]*Organ`, wrap it with `ByName` or `ByWeight` and pass it to `sort.Sort`:

```
s := []*Organ{
    {"brain", 1340},
    {"heart", 290},
    {"liver", 1494},
    {"pancreas", 131},
    {"spleen", 162},
}

sort.Sort(ByWeight{s})
printOrgans("Organs by weight", s)

sort.Sort(ByName{s})
printOrgans("Organs by name", s)
```

[Run](#)

Another wrapper

The `Reverse` function takes a `sort.Interface` and returns a `sort.Interface` with an inverted `Less` method:

```
func Reverse(data sort.Interface) sort.Interface {
    return &reverse{data}
}

type reverse struct{ sort.Interface }

func (r reverse) Less(i, j int) bool {
    return r.Interface.Less(j, i)
}
```

To sort the organs in descending order, compose our sort types with `Reverse`:

```
sort.Sort(Reverse(ByWeight{s}))
printOrgans("Organs by weight (descending)", s)

sort.Sort(Reverse(ByName{s}))
printOrgans("Organs by name (descending)", s)
```

[Run](#)

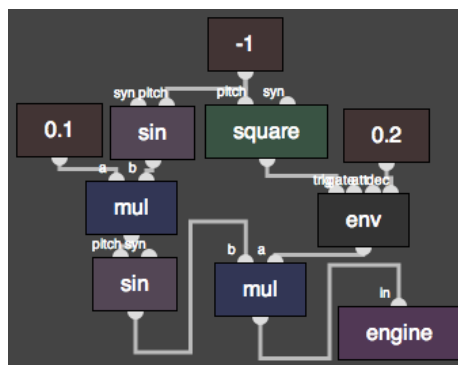
Interfaces: why they work

These are not just cool tricks.

This is how we structure programs in Go.

Interfaces: Sigourney

Sigourney is a modular audio synthesizer I wrote in Go.



Audio is generated by a chain of Processors:

```
type Processor interface {  
    Process(buffer []Sample)  
}
```

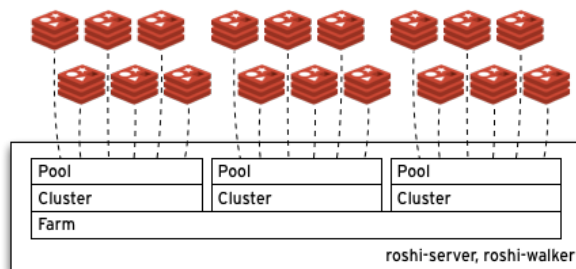
(github.com/nf/sigourney (<https://github.com/nf/sigourney>))

Interfaces: Roshi

Roshi is a time-series event store written by Peter Bourgon. It provides this API:

```
Insert(key, timestamp, value)
Delete(key, timestamp, value)
Select(key, offset, limit) []TimestampValue
```

The same API is implemented by the `farm` and `cluster` parts of the system.



An elegant design that exhibits composition.

(github.com/soundcloud/roshi (<https://github.com/soundcloud/roshi>))

Interfaces: why they work (continued)

Interfaces are *the* generic programming mechanism.

This gives all Go code a familiar shape.

Less is more.

Interfaces: why they work (continued)

It's all about composition.

Interfaces—by design and convention—encourage us to write composable code.

Interfaces: why they work (continued)

Interfaces types are just types
and interface values are just values.

They are orthogonal to the rest of the language.

Interfaces: why they work (continued)

Interfaces separate data from behavior. (Classes conflate them.)

```
type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

Interfaces: what I learned

Think about composition.

Better to have many small simple things than one big complex thing.

Also: what I thought of as small is pretty big.

Some repetition in the small is okay when it benefits "the large".

Concurrency

Concurrency: first impressions

My first exposure to concurrency was in C, Java, and Python.
Later: event-driven models in Python and JavaScript.

When I saw Go I saw:

"The performance of an event-driven model without callback hell."

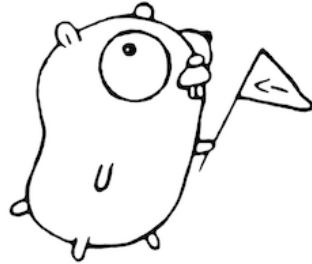
But I had questions: "Why can't I wait on or kill a goroutine?"

Concurrency: the Go way

Goroutines provide concurrent execution.

Channels express the communication and synchronization of independent processes.

Select enables computation on channel operations.



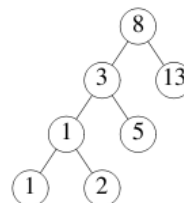
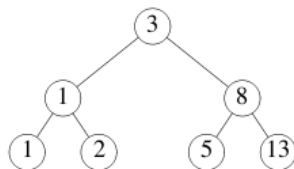
A concurrency example

The binary tree comparison exercise from the Go Tour.

"Implement a function

```
func Same(t1, t2 *tree.Tree) bool
```

that compares the contents of two binary trees."



Walking a tree

```
type Tree struct {  
    Left, Right *Tree  
    Value int  
}
```

A simple depth-first tree traversal:

```
func Walk(t *tree.Tree) {  
    if t.Left != nil {  
        Walk(t.Left)  
    }  
    fmt.Println(t.Value)  
    if t.Right != nil {  
        Walk(t.Right)  
    }  
}  
  
func main() {  
    Walk(tree.New(1))  
}
```

[Run](#)

Comparing trees (1/2)

A concurrent walker:

```
func Walk(root *tree.Tree) chan int {  
    ch := make(chan int)  
    go func() {  
        walk(root, ch)  
        close(ch)  
    }()  
    return ch  
}  
  
func walk(t *tree.Tree, ch chan int) {  
    if t.Left != nil {  
        walk(t.Left, ch)  
    }  
    ch <- t.Value  
    if t.Right != nil {  
        walk(t.Right, ch)  
    }  
}
```

Comparing trees (2/2)

Walking two trees concurrently:

```
func Same(t1, t2 *tree.Tree) bool {
    w1, w2 := Walk(t1), Walk(t2)
    for {
        v1, ok1 := <-w1
        v2, ok2 := <-w2
        if v1 != v2 || ok1 != ok2 {
            return false
        }
        if !ok1 {
            return true
        }
    }
}

func main() {
    fmt.Println(Same(tree.New(3), tree.New(3)))
    fmt.Println(Same(tree.New(1), tree.New(2)))
}
```

[Run](#)

Comparing trees without channels (1/3)

```
func Same(t1, t2 *tree.Tree) bool {
    w1, w2 := Walk(t1), Walk(t2)
    for {
        v1, ok1 := w1.Next()
        v2, ok2 := w2.Next()
        if v1 != v2 || ok1 != ok2 {
            return false
        }
        if !ok1 {
            return true
        }
    }
}
```

The Walk function has nearly the same signature:

```
func Walk(root *tree.Tree) *Walker {
```

```
func (w *Walker) Next() (int, bool) {
```

(We call Next instead of the channel receive.)

Comparing trees without channels (2/3)

But the implementation is much more complex:

```
func Walk(root *tree.Tree) *Walker {
    return &Walker{stack: []*frame{{t: root}}}
}

type Walker struct {
    stack []*frame
}

type frame struct {
    t *tree.Tree
    pc int
}

func (w *Walker) Next() (int, bool) {
    if len(w.stack) == 0 {
        return 0, false
    }

    // continued next slide ...
}
```

Comparing trees without channels (3/3)

```
f := w.stack[len(w.stack)-1]
if f.pc == 0 {
    f.pc++
    if l := f.t.Left; l != nil {
        w.stack = append(w.stack, &frame{t: l})
        return w.Next()
    }
}
if f.pc == 1 {
    f.pc++
    return f.t.Value, true
}
if f.pc == 2 {
    f.pc++
    if r := f.t.Right; r != nil {
        w.stack = append(w.stack, &frame{t: r})
        return w.Next()
    }
}
w.stack = w.stack[:len(w.stack)-1]
return w.Next()
}
```

Another look at the channel version

```
func Walk(root *tree.Tree) chan int {
    ch := make(chan int)
    go func() {
        walk(root, ch)
        close(ch)
    }()
    return ch
}

func walk(t *tree.Tree, ch chan int) {
    if t.Left != nil {
        walk(t.Left, ch)
    }
    ch <- t.Value
    if t.Right != nil {
        walk(t.Right, ch)
    }
}
```

But there's a problem: when an inequality is found, a goroutine might be left blocked sending to ch.

Stopping early

Add a quit channel to the walker so we can stop it mid-stride.

```
func Walk(root *tree.Tree, quit chan struct{}) chan int {
    ch := make(chan int)
    go func() {
        walk(root, ch, quit)
        close(ch)
    }()
    return ch
}

func walk(t *tree.Tree, ch chan int, quit chan struct{}) {
    if t.Left != nil {
        walk(t.Left, ch, quit)
    }
    select {
    case ch <- t.Value:
    case <-quit:
        return
    }
    if t.Right != nil {
        walk(t.Right, ch, quit)
    }
}
```

Stopping early (continued)

Create a quit channel and pass it to each walker.

By closing quit when the Same exits, any running walkers are terminated.

```
func Same(t1, t2 *tree.Tree) bool {  
    quit := make(chan struct{})  
    defer close(quit)  
    w1, w2 := Walk(t1, quit), Walk(t2, quit)  
    for {  
        v1, ok1 := <-w1  
        v2, ok2 := <-w2  
        if v1 != v2 || ok1 != ok2 {  
            return false  
        }  
        if !ok1 {  
            return true  
        }  
    }  
}
```

Why not just kill the goroutines?

Goroutines are invisible to Go code. They can't be killed or waited on.

You have to build that yourself.

There's a reason:

As soon as Go code knows in which thread it runs you get thread-locality.

Thread-locality defeats the concurrency model.

Concurrency: why it works

The model makes concurrent code easy to read and write.
(Makes concurrency is **accessible**.)

This encourages the decomposition of independent computations.

Concurrency: why it works (continued)

The simplicity of the concurrency model makes it flexible.

Channels are just values; they fit right into the type system.

Goroutines are invisible to Go code; this gives you concurrency anywhere.

Less is more.

Concurrency: what I learned

Concurrency is not just for doing more things faster.

It's for writing better code.

Syntax

Syntax: first impressions

At first, Go syntax felt a bit inflexible and verbose.

It affords few of the conveniences to which I was accustomed.

For instance:

- No getters/setters on fields.
- No map/filter/reduce/zip.
- No optional arguments.

Syntax: the Go way

Favor readability above all.

Offer enough sugar to be productive, but not too much.

Getters and setters (or "properties")

Getters and setters turn assignments and reads into function calls.
This leads to surprising hidden behavior.

In Go, just write (and call) the methods.

The control flow cannot be obscured.

Map/filter/reduce/zip

Map/filter/reduce/zip are useful in Python.

```
a = [1, 2, 3, 4]
b = map(lambda x: x+1, a)
```

In Go, you just write the loops.

```
a := []int{1, 2, 3, 4}
b := make([]int, len(a))
for i, x := range a {
    b[i] = x+1
}
```

This is a little more verbose,
but makes the performance characteristics obvious.

It's easy code to write, and you get more control.

Optional arguments

Go functions can't have optional arguments.

Instead, use variations of the function:

```
func NewWriter(w io.Writer) *Writer
func NewWriterLevel(w io.Writer, level int) (*Writer, error)
```

Or an options struct:

```
func New(o *Options) (*Jar, error)

type Options struct {
    PublicSuffixList PublicSuffixList
}
```

Or a variadic list of options.

Create many small simple things, not one big complex thing.

Syntax: why it works

The language resists convoluted code.

With obvious control flow, it's easy to navigate unfamiliar code.

Instead we create more small things that are easy to document and understand.

So Go code is easy to read.

(And with gofmt, it's easy to write readable code.)

Syntax: what I learned

I was often too clever for my own good.

I appreciate the consistency, clarity, and *transparency* of Go code.

I sometimes miss the conveniences, but rarely.

Error handling

Error handling: first impressions

I had previously used exceptions to handle errors.

Go's error handling model felt verbose by comparison.

I was immediately tired of typing this:

```
if err != nil {  
    return err  
}
```

Error handling: the Go way

Go codifies errors with the built-in error interface:

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

Error values are used just like any other value.

```
func doSomething() error  
  
err := doSomething()  
if err != nil {  
    log.Println("An error occurred:", err)  
}
```

Error handling code is just code.

(Started as a convention (`os.Error`). We made it built in for Go 1.)

Error handling: why it works

Error handling is important.

Go makes error handling as important as any other code.

Error handling: why it works (continued)

Errors are just values; they fit easily into the rest of the language (interfaces, channels, and so on).

Result: Go code handles errors correctly and elegantly.

Error handling: why it works (continued)

We use the same language for errors as everything else.

Lack of hidden control flow (throw/try/catch/finally) improves readability.

Less is more.

Error handling: what I learned

To write good code we must think about errors.

Exceptions make it easy to avoid thinking about errors.
(Errors shouldn't be "exceptional!")

Go encourages us to consider every error condition.

My Go programs are far more robust than my programs in other languages.

I don't miss exceptions at all.

Packages

Packages: first impressions

I found the capital-letter-visibility rule weird;
"Let me use my own naming scheme!"

I didn't like "package per directory";
"Let me use my own structure!"

I was disappointed by lack of monkey patching.

Packages: the Go way

Go packages are a name space for types, functions, variables, and constants.

Visibility

Visibility is at the package level.

Names are "exported" when they begin with a capital letter.

```
package zip

func NewReader(r io.ReaderAt, size int64) (*Reader, error) // exported

type Reader struct {    // exported
    File    []*File    // exported
    Comment string    // exported
    r       io.ReaderAt // unexported
}

func (f *File) Open() (rc io.ReadCloser, err error)    // exported

func (f *File) findBodyOffset() (int64, error)        // unexported

func readDirectoryHeader(f *File, r io.Reader) error  // unexported
```

Good for readability: easy to see whether a name is part of the public interface.

Good for design: couples naming decisions with interface decisions.

Package structure

Packages can be spread across multiple files.

Permits shared private implementation and informal code organization.

Packages files must live in a directory unique to the package.

The path to that directory determines the package's import path.

The build system locates dependencies from the source alone.

"Monkey patching"

Go forbids modifying package declarations from outside the package.

But we can get similar behavior using global variables:

```
package flag

var Usage = func() {
    fmt.Fprintf(os.Stderr, "Usage of %s:\n", os.Args[0])
    PrintDefaults()
}
```

Or registration functions:

```
package http

func Handle(pattern string, handler Handler)
```

This gives the flexibility of monkey patching but on the package author's terms.

(This depends on Go's initialization semantics.)

Packages: why they work

The loose organization of packages lets us write and refactor code quickly.

But packages encourage the programmer to consider the public interface.

This leads to good names and simpler interfaces.

With the source as the single source of truth,
there are no makefiles to get out of sync.

(This design enables great tools like godoc.org and goimports.)

Predictable semantics make packages easy to read, understand, and use.

Packages: what I learned

Go's package system taught me to prioritize the consumer of my code.
(Even if that consumer is me.)

It also stopped me from doing gross stuff.

Packages are rigid where it matters, and loose where it doesn't.
It just feels right.

Probably my favorite part of the language.

Documentation

Documentation: first impressions

Godoc reads documentation from Go source code, like pydoc or javadoc.

But unlike those two, it doesn't support complex formatting or other meta data.
Why?

Documentation: the Go way

Godoc comments precede the declaration of an exported identifier:

```
// Join concatenates the elements of a to create a single string.  
// The separator string sep is placed between elements in the resulting string.  
func Join(a []string, sep string) string {
```

It extracts the comments and presents them:

```
$ godoc strings Join  
func Join(a []string, sep string) string  
    Join concatenates the elements of a to create a single string. The  
    separator string sep is placed between elements in the resulting string.
```

Also integrated with the testing framework to provide testable example functions.

```
func ExampleJoin() {  
    s := []string{"foo", "bar", "baz"}  
    fmt.Println(strings.Join(s, " "))  
    // Output: foo, bar, baz  
}
```

Documentation: the Go way (continued)

func Join

```
func Join(a []string, sep string) string
```

Join concatenates the elements of a to create a single string. The separator string sep is placed between elements in the resulting string.

▼ Example

```
package main  
  
import (  
    "fmt"  
    "strings"  
)  
  
func main() {  
    s := []string{"foo", "bar", "baz"}  
    fmt.Println(strings.Join(s, " "))  
}
```

foo, bar, baz

Program exited.

[Run](#)[Format](#)[Share](#)

Documentation: why it works

Godoc wants you to write good comments, so the source looks great:

```
// ValidMove reports whether the specified move is valid.  
func ValidMove(from, to Position) bool
```

Javadoc just wants to produce pretty documentation, so the source is hideous:

```
/**  
 * Validates a chess move.  
 *  
 * @param fromPos position from which a piece is being moved  
 * @param toPos   position to which a piece is being moved  
 * @return        true if the move is valid, otherwise false  
 */  
boolean isValidMove(Position fromPos, Position toPos)
```

(Also a grep for "ValidMove" will return the first line of documentation.)

Documentation: what I learned

Godoc taught me to write documentation *as I code*.

Writing documentation *improves the code* I write.

More

There are many more examples.

The overriding theme:

- At first, something seemed weird or lacking.
- I realized it was a design decision.

Those decisions make the language—and Go code—better.

Sometimes you have to live with the language a while to see it.

Lessons

Code is communication

Be articulate:

- Choose good names.
- Design simple interfaces.
- Write precise documentation.
- Don't be too clever.

Less is exponentially more

New features can weaken existing features.

Features multiply complexity.

Complexity defeats orthogonality.

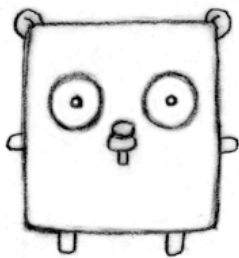
Orthogonality is vital: it enables composition.

Composition is key

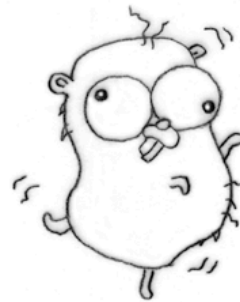
Don't solve problems by building *a* thing.

Instead, combine simple tools and compose them.

Design good interfaces



Don't over-specify.



Don't under-specify.

Find the sweet spot.

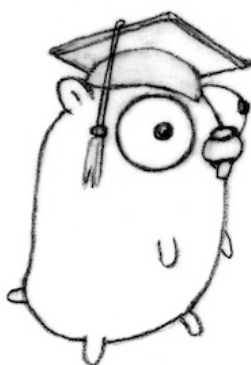
Simplicity is hard

Invest the time to find the simple solution.

Go's effect on me

These lessons were all things I already "knew".

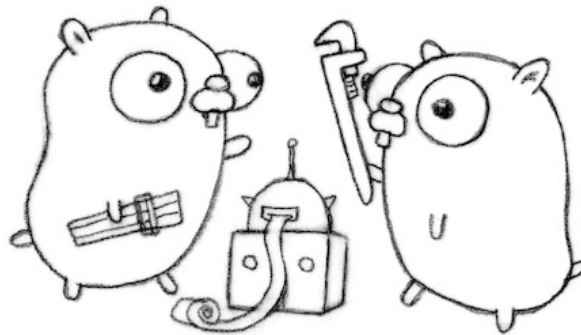
Go helped me internalize them.



Go made me a better programmer.

A message for gophers everywhere

Let's build small, simple, and beautiful things together.



Thank you

Andrew Gerrand
Google, Inc.

[@enneff](http://twitter.com/enneff) (<http://twitter.com/enneff>)

adg@golang.org (<mailto:adg@golang.org>)