# The Go features I can't live without

## Golang Brno meetup #1
## 17 May 2016

Rodolfo Carvalho
Red Hat

# First Golang Brno meeting

Survey time!

- Who comes from other cities/countries?
- Who never wrote a single line of Go code?
- Who writes Go code frequently?

## Me

Go since ???

- Jul 2011 chat message via ? (release.r58.1 / weekly.2011-07-19)
- Sep 2011 via Racket mailing list
- Jan 2013 via email, then 什么词 shenmeci & DAWGo (go1.0)
- Oct 2013 Golang KRK
- ...
- Apr 2015 OpenShift developer

## This talk

- May serve as a brief intro to Go (see the Tour of Go!)
- Features that I personally enjoy and miss when I code in other languages

1

## Language simplicity

- Less is more
- Fits your brain
- Easy to learn
- Easy to read
- Suitable for larger teams
- Other languages compete on feature set

Rob Pike explains it better at youtu.be/rFejpH_tAHM

## Highlights

Too many features means time wasted choosing which ones to use.

If the language is complicated:

- You must understand more things to read and work on the code.
- You must understand more things to debug and fix it.

Tradeoff: More fun to write, or less work to maintain?

- Go is boring!

2

# Single dispatch

Julia:

```
A Summary of Features
- Multiple dispatch: providing ability to define function behavior across many
  combinations of argument types
- ...
```

Go has single dispatch. Simpler to reason about.

And no operator overloading, no "magic", ... and no generics.

## Example in Julia

Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most powerful and central feature of the Julia language. Core operations typically have dozens of methods:

```
julia> methods(+)
# 139 methods for generic function "+":
+(x::Bool) at bool.jl:33
+(x::Bool,y::Bool) at bool.jl:36
+(y::AbstractFloat,x::Bool) at bool.jl:46
+(x::Int64,y::Int64) at int.jl:14
+(x::Int8,y::Int8) at int.jl:14
+(x::UInt8,y::UInt8) at int.jl:14
+(x::Int16,y::Int16) at int.jl:14
+(x::UInt16,y::UInt16) at int.jl:14
+(x::Int32,y::Int32) at int.jl:14
+(x::UInt32,y::UInt32) at int.jl:14
+(x::UInt64,y::UInt64) at int.jl:14
+(x::Int128,y::Int128) at int.jl:14
...
```

docs.julialang.org/en/release-0.4/manual/methods/#man-methods

# Example in Go

```
func (d *downloader) Download(url *url.URL, targetFile string) (*api.SourceInfo, error) {
    schemeReader := d.schemeReaders[url.Scheme]
    info := &api.SourceInfo{}
    if schemeReader == nil {
        glog.Errorf("No URL handler found for %s", url.String())
        return nil, errors.NewURLHandlerError(url.String())
    }

    reader, err := schemeReader.Read(url)
    if err != nil {
        return nil, err
    }
    defer reader.Close()
```

Things are what they look like, easy to reason about code.
Also, grep friendly.

3

## Capitalization is meaningful

An identifier may be exported to permit access to it from another package. An identifier is exported if both:

1. the first character of the identifier's name is a Unicode upper case letter; and
2. the identifier is declared in the package block or it is a field name or method name.

All other identifiers are not exported.

# Example

```
func (d *downloader) Download(url *url.URL, targetFile string) (*api.SourceInfo, error) {
    schemeReader := d.schemeReaders[url.Scheme]
    info := &api.SourceInfo{}
    if schemeReader == nil {
        glog.Errorf("No URL handler found for %s", url.String())
        return nil, errors.NewURLHandlerError(url.String())
    }

    reader, err := schemeReader.Read(url)
    if err != nil {
        return nil, err
    }
    defer reader.Close()
```

4

## gofmt

- Go is recognized for good tooling
- Executable, automatic coding standard (Python's PEP8 on steroids)

Also:

- go fix
- go vet
- and many linters

5

# godoc

- Extracts, generates and serves documentation

```
$ go doc fmt Println
func Println(a ...interface{}) (n int, err error)
    Println formats using the default formats for its operands and writes to
    standard output. Spaces are always added between operands and a newline is
    appended. It returns the number of bytes written and any write error
    encountered.
```

- Docs live near code

```
// Println formats using the default formats for its operands and writes to standard output.
// Spaces are always added between operands and a newline is appended.
// It returns the number of bytes written and any write error encountered.
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

- Culture
- godoc.org

6

# No exceptions!

- No exceptions, explicit error handling using all of the power of the language
- No especial syntax / constructs to learn
- Errors are values!

```
_, err = fd.Write(p0[a:b])
if err != nil {
    return err
}
_, err = fd.Write(p1[c:d])
if err != nil {
    return err
}
_, err = fd.Write(p2[e:f])
if err != nil {
    return err
}
// and so on
```

- Playing fair... do you really think and write code to handle exceptional cases?

# Programming your error handling

```
type errWriter struct {
    w   io.Writer
    err error
}
func (ew *errWriter) write(buf []byte) {
    if ew.err != nil {
        return
    }
    _, ew.err = ew.w.Write(buf)
}
```

Use it:

```
ew := &errWriter{w: fd}
ew.write(p0[a:b])
ew.write(p1[c:d])
ew.write(p2[e:f])
// and so on
if ew.err != nil {
    return ew.err
}
```

7

## Table-driven tests

Because there are no exceptions and because errors are values, it is easy to write table-driven tests.

Define tests:

```
type atoi64Test struct {
    in  string
    out int64
    err error
}

var atoi64tests = []atoi64Test{
    {"", 0, ErrSyntax},
    {"0", 0, nil},
    // ...
    {"9223372036854775809", 1<<63 - 1, ErrRange},
    {"-9223372036854775809", -1 << 63, ErrRange},
}
```

## Test them all

```
func TestParseInt64(t *testing.T) {
    for i := range atoi64tests {
        test := &atoi64tests[i]
        out, err := ParseInt(test.in, 10, 64)
        if test.out != out || !reflect.DeepEqual(test.err, err) {
            t.Errorf("Atoi64(%q) = %v, %v want %v, %v",
                test.in, out, err, test.out, test.err)
        }
    }
}
```

Even if a single test case fails, all other tests can still be run.

There are no *assertions*, no *exceptions* to control the test flow, and no need for the "single assert per test method" rule.

## Interfaces

Go interfaces enable post-facto abstraction.

- In Python, you can't write it down and type-check, you just duck type
- In Java, you must declare *a priori*: 3rd-party code cannot implement your interfaces
- Implicitly satisfiable interfaces

Frequently used: `io.Reader`, `io.Writer`, `fmt.Stringer`, `interface{}`

Example:

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

# The bigger the interface, the weaker the abstraction

Keep them small, and compose!

```
type ReadWriter interface {
    Reader
    Writer
}
```

# To implement an interface

Implement methods with the appropriate signatures:

```
type File struct {
    Name string
}

func (f *File) Read(p []byte) (n int, err error) {
    return len(p), nil
}

func (f *File) Write(p []byte) (n int, err error) {
    return len(p), nil
}

func (f *File) String() string {
    return f.Name
}
```

Nothing more.

## To define your own interface

Declare the method signatures:

```
type Greeter interface {
    Greet(string) error
}
```

# Swiss-knife interface

go-review.googlesource.com/#/c/20763/

```
// go/src/net/http/request.go
func (l *maxBytesReader) tooLarge() (n int, err error) {
    if !l.stopped {
        l.stopped = true

        // The server code and client code both use
        // maxBytesReader. This "requestTooLarge" check is
        // only used by the server code. To prevent binaries
        // which only using the HTTP Client code (such as
        // cmd/go) from also linking in the HTTP server, don't
        // use a static type assertion to the server
        // "*response" type. Check this interface instead:
        type requestTooLarger interface {
            requestTooLarge()
        }
        if res, ok := l.w.(requestTooLarger); ok {
            res.requestTooLarge()
        }
    }
    return 0, errors.New("http: request body too large")
}
```

# Recap

1. **Simplicity**
2. Single dispatch
3. Capitalization
4. gofmt
5. godoc
6. No exceptions
7. Table-Driven Tests
8. Interfaces

## Next time

9. First-class functions
10. Fully qualified imports
11. Static typing with type inference
12. Speed
13. Metaprogramming
14. Static linking
15. Cross-compilation
16. Go Proverbs

# I want to code!

- Coding Dojo Brno
- Every Wednesday, 18:00 - 20:00

github.com/dojo-brno/src

# Thank you

Rodolfo Carvalho
Red Hat
rhcarvalho@gmail.com
http://rodolfocarvalho.net