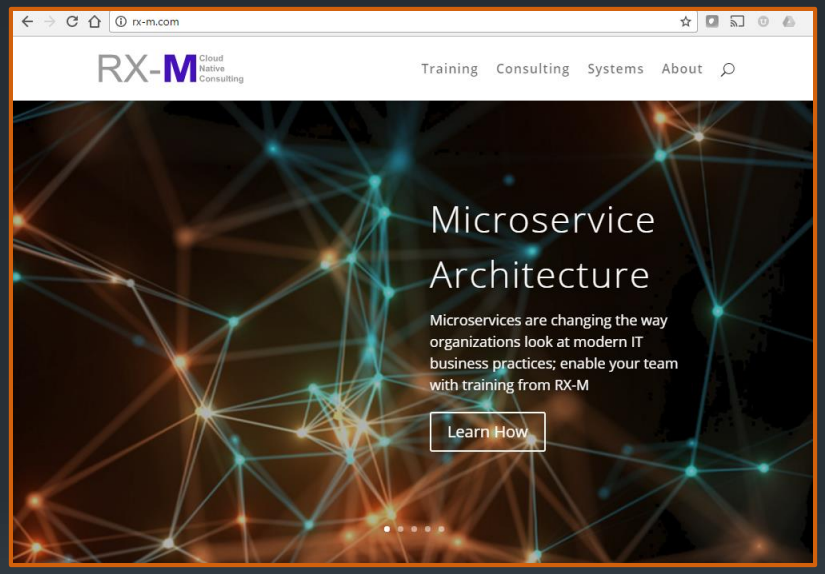
RX-M Cloud Native Consulting

Go

Foundation

RX-M Cloud Native Training

- Microservice Oriented
  - Microservices Foundation [2 Day]
  - Building Microservices with Go [3 Day]
  - Java Microservices with Spring [2 Day]
- Container Packaged
  - Docker Foundation [3 Day]
  - Docker Advanced [2 Day]
  - Rocket [2 Day]
- Dynamically Managed
  - Docker Orchestration (Compose/Swarm) [2 Day]
  - Kubernetes Foundation [2 Day]
  - Kubernetes Advanced [2 Day]
  - Apache Mesos Foundation [2 Day]
  - Marathon [2 Day]
  - Nomad [2 Day]



RX-M Cloud Native Consulting

Overview

Day One

1. Go Overview
2. Syntax and Flow Control
3. Data Types
4. Program Construction

Day Two

1. User Defined Types
2. Functions
3. Methods and Interfaces
4. Error Handling

Day Three

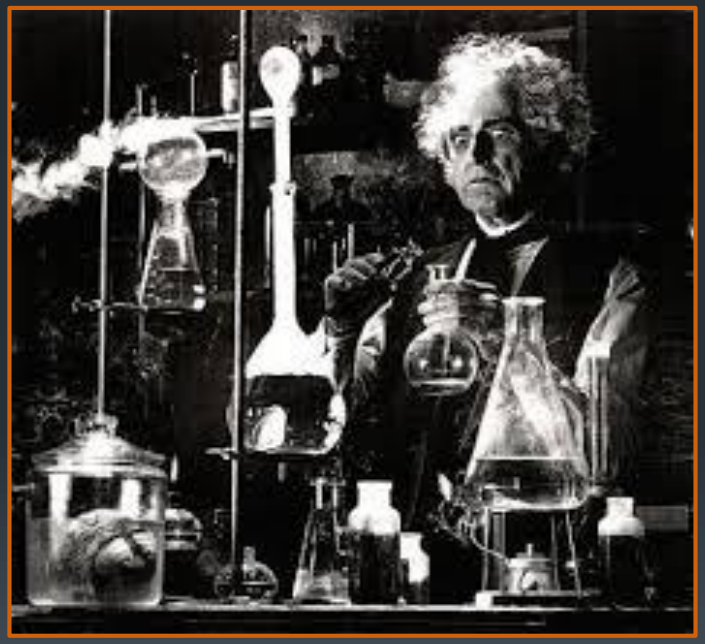
1. CSP
2. Standard Library
3. Testing
4. Debugging

Administrative Info

- Length: 2 Days
- Format: Lecture/Labs/Discussion
- Schedule: 9:30AM – 5:30PM
  15 minute break, AM & PM
  1 hour lunch at noon
  Lab time at the end of each AM and PM session
- Location: Fire exits, Restrooms, Security, other matters
- Attendees: Name/Role/Experience/Goals for the Course

Lecture and Lab

- Our Goals in this class are two fold:
  1. Familiarize you with the concepts
     - This is the primary purpose of the lecture/discussion sessions
  2. Give you practical experience
     - This is the primary purpose of the labs



1: Go Overview

Objectives

- Describe the reasons Go was created
- Explain the design philosophy of Go
- Contrast Go with other languages
- Understand Go basics
- Explore the Go eco system
- Locate Go documentation and community resources

Go Defined

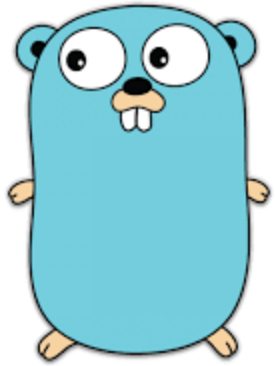
- Go is an open source programming language that makes it easy to build simple, reliable, and efficient software
  -- golang.org

- Often referred to as golang
- Created at Google in 2007
  - Used in some of Google's production systems

- Two major implementations exist
  - Tools are accessed through a single command called go that has a number of subcommands (run, build, get, etc.)
  - gc – Google's Go compiler
    - Supports Linux, OS X, Windows, various BSD/Unix, and mobile devices
    - Self hosted as of v1.5 (the Go compiler is written in Go)
    - The original and principle Go compiler
  - gccgo – A GCC Go frontend
    - Compiles Go with the Gnu C/C++ compiler gcc
    - gccgo tooling has traditionally been slower to compile but faster at runtime due to vast set of optimizations available in gcc
    - Having two compiler implementations ensures the Go spec is concise and explicit

- Go Features
  - Compiled
  - Statically typed
  - C syntax
  - Garbage collection
  - Memory safety features
  - CSP-style concurrency
  - Native UTF-8 strings

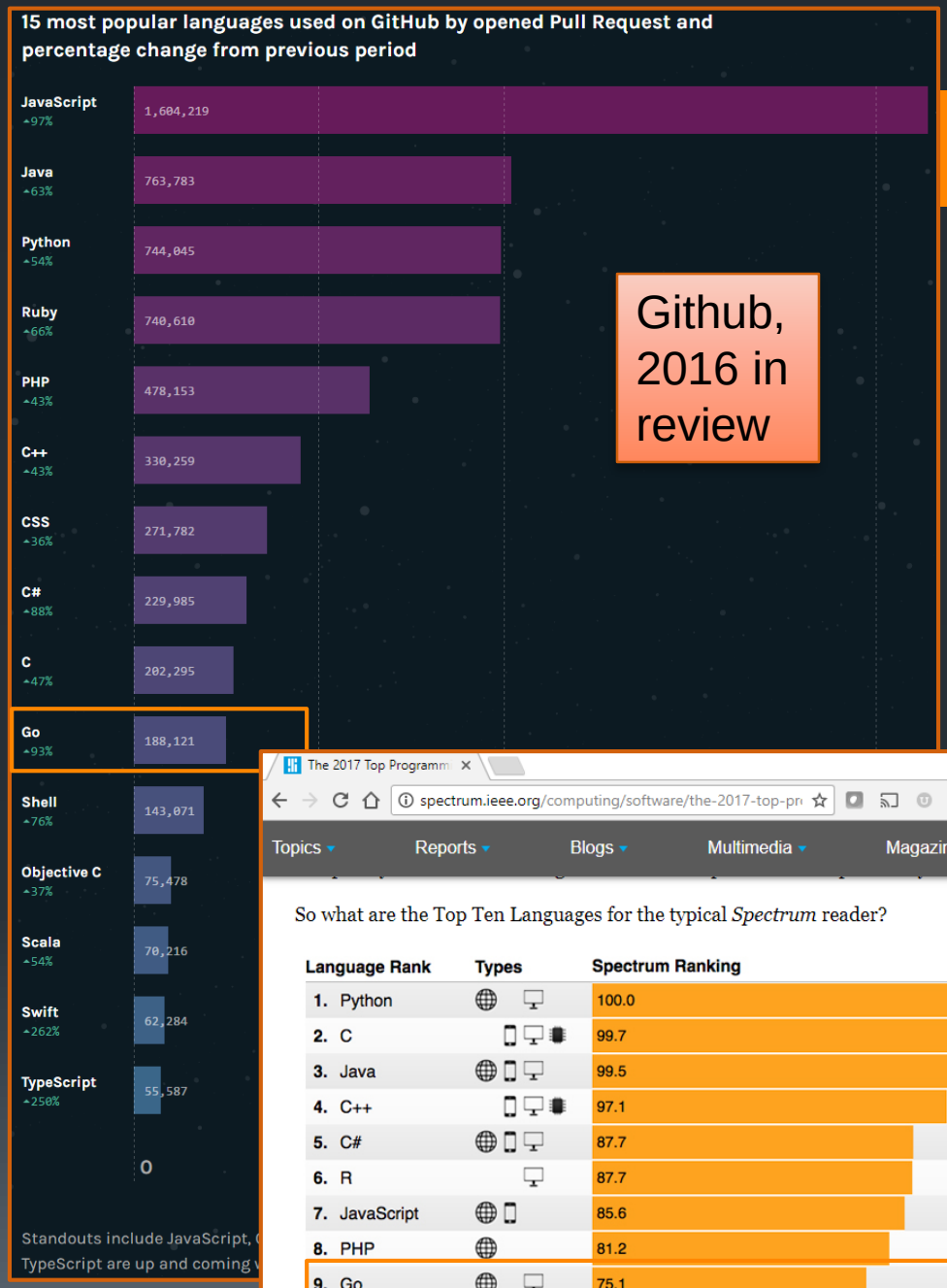
> *The key point here is our programmers are Googlers, they're not researchers. They're typically, fairly young, fresh out of school, probably learned Java, maybe learned C or C++, probably learned Python. They're not capable of understanding a brilliant language but we want to use them to build good software. So, the language that we give them has to be easy for them to understand and easy to adopt.*
>
> *-- Rob Pike*



The Go pher

Motivation for Go

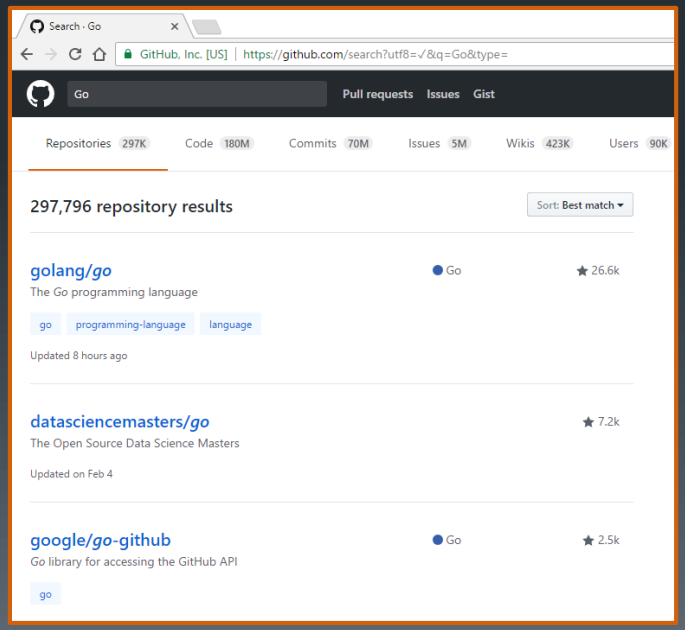
- Don't we have enough programming languages already?
  - Some don't think so
- Go originated as an experiment by Google engineers
  - Robert Griesemer
  - Rob Pike
  - Ken Thompson
- The goal:
  - To design a new programming language that would resolve common criticisms of other languages while maintaining their positive characteristics
- Go included the following features:
  - Static typing
  - Scalable to large systems (like Java and C++)
  - Highly productive and readable
  - No proliferation of mandatory keywords
  - Avoiding repetition
    - "light on the page" like dynamic languages
  - No need for IDEs (integrated development environments), but support for them
  - Support for networking
  - Support for multiprocessing
- In interviews all three designers cited their shared dislike of C++'s complexity as a primary motivation for designing a new language

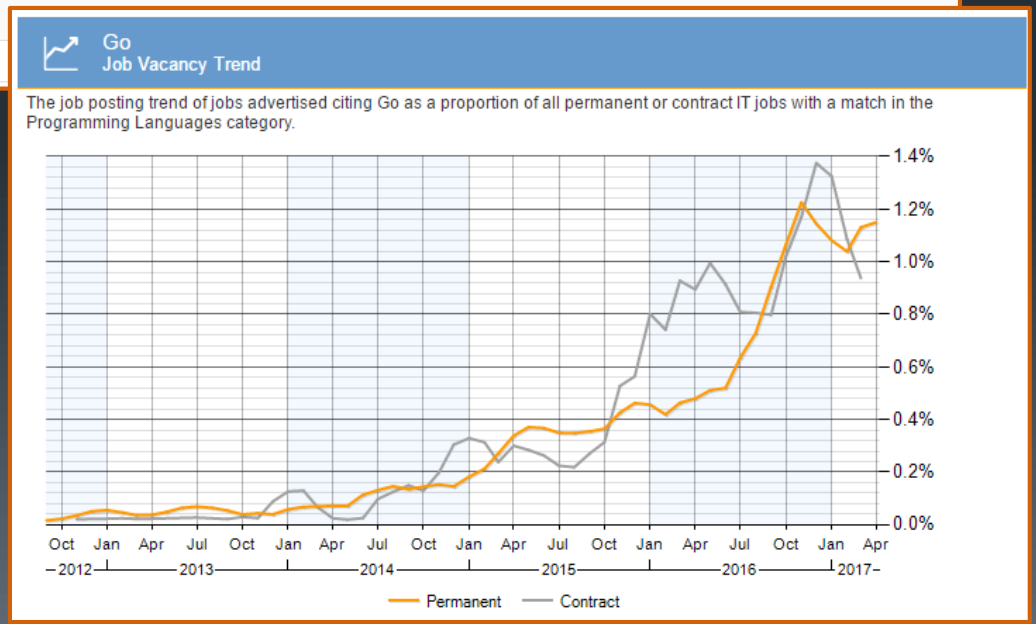


Growing Popularity

- Per the Tiobe index, Go is the 17th most popular programming language as of March 2017
  - Up from 48th the year before
  - Go had the largest place gain of any language in 2009, the year of its introduction
- Nearly 300k repository results match a search on GitHub for Go
- Demand for Go programmers has been growing significantly

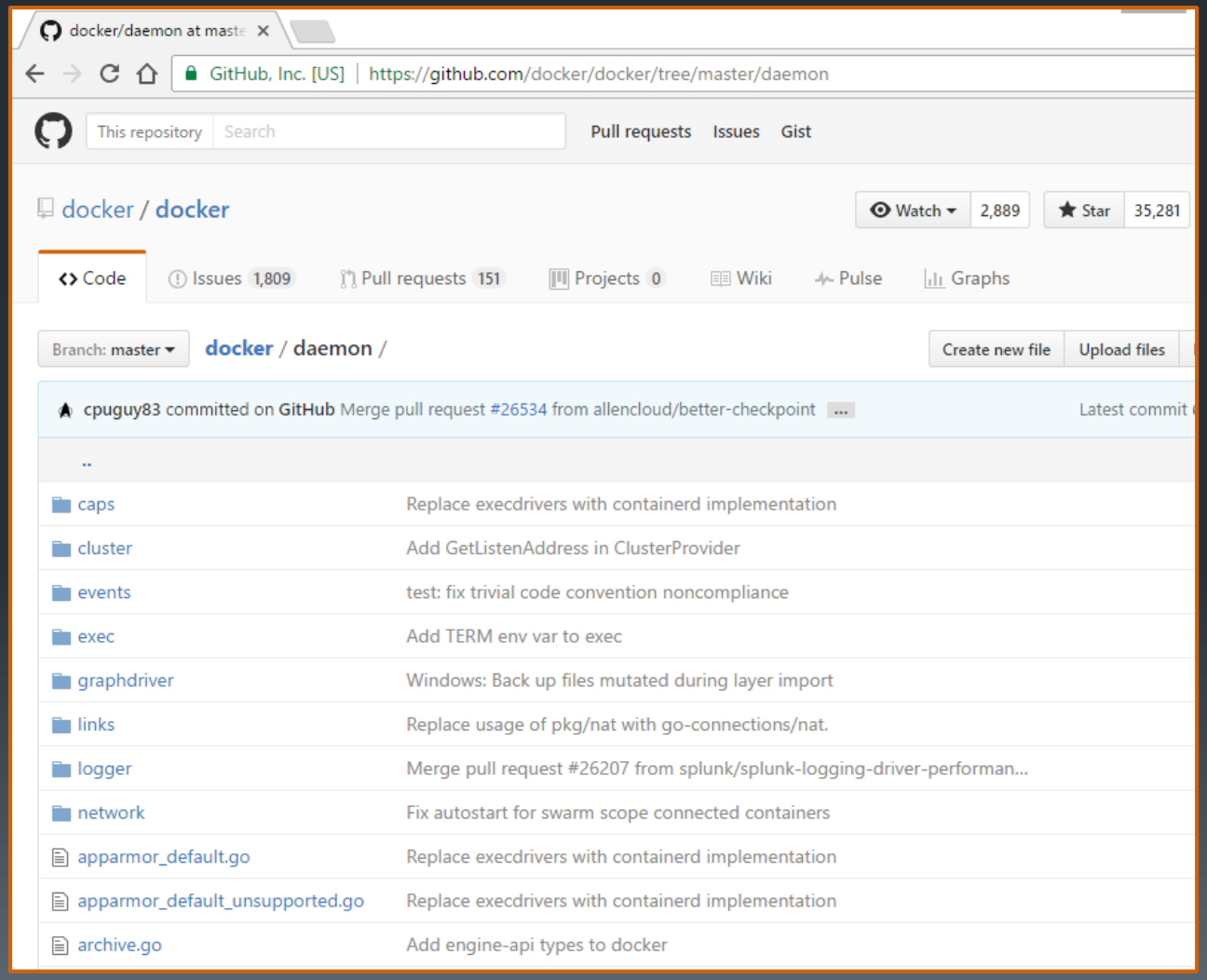
Mar 2017	Mar 2016	Change	Programming Language	Ratings	Change
1	1		Java	16.384%	-4.14%
2	2		C	7.742%	-6.86%
3	3		C++	5.184%	-1.54%
4	4		C#	4.409%	+0.14%
5	5		Python	3.919%	-0.34%
6	7	^	Visual Basic .NET	3.174%	+0.61%
7	6	v	PHP	3.009%	+0.24%
8	8		JavaScript	2.667%	+0.33%
9	11	^	Delphi/Object Pascal	2.544%	+0.54%
10	14	^^	Swift	2.268%	+0.68%
11	9	v	Perl	2.261%	+0.01%
12	10	v	Ruby	2.254%	+0.02%
13	12	v	Assembly language	2.232%	+0.39%
14	16	^	R	2.016%	+0.73%
15	13	v	Visual Basic	2.008%	+0.33%
16	15	v	Objective-C	1.997%	+0.54%
17	48	^^	Go	1.982%	+1.78%
18	18		MATLAB	1.854%	+0.66%
19					
20					





The Language of Container Tech

- While Go is used in many areas, container technology is dominated by Go
- Tools written in Go:
  - Docker
  - OCI
    - Rocket
    - Containerd
  - Kubernetes
  - Etcd
  - Consul
  - Nats
  - InfluxDB
  - cAdvisor
  - Prometheus
  - Many more



Go Versions

- 2017-05-24 go1.8.3
- 2017-05-23 go1.8.2
- 2017-04-07 go1.8.1
- 2017-02-16 go1.8
- 2017-01-26 go1.7.5
- 2016-12-01 go1.7.4 & go1.6.4
- 2016-10-19 go1.7.3
- 2016-10-17 go1.7.2
- 2016-09-07 go1.7.1
- 2016-08-15 go1.7
- 2016-07-18 go1.6.3
- 2016-04-19 go1.6.2
- 2016-04-11 go1.6.1 & go1.5.4
- 2016-02-17 go1.6
- 2016-01-13 go1.5.3
- 2015-12-02 go1.5.2
- 2015-09-22 go1.4.3
- 2015-09-08 go1.5.1
- 2015-08-18 go1.5 (Go becomes self hosting)
- 2015-02-17 go1.4.2
- 2015-01-15 go1.4.1
- 2014-12-10 go1.4
- 2014-09-30 go1.3.3
- 2014-09-25 go1.3.2
- 2014-08-12 go1.3.1
- 2014-06-18 go1.3
- 2014-05-05 go1.2.2
- 2014-05-02 go1.2.1
- 2013-11-28 go1.2
- 2013-08-12 go1.1.2
- 2013-06-12 go1.1.1
- 2013-05-13 go1.1
- 2012-09-21 go1.0.3
- 2012-06-13 go1.0.2
- 201204-26 go1.0.1
- 2012-03-28 go1

- Prior to Go v1 maintainers released weekly builds
- In 2011 releases were designated rXX
  - 2011-10-17 release.r58.2 & release.r60.3
  - 2011-10-05 release.r60.2
  - 2011-09-18 release.r60.1
  - 2011-09-07 release.r60
  - 2011-07-31 release.r59
  - 2011-07-12 release.r58.1
  - 2011-06-29 release.r58
  - 2011-06-15 release.r57.2
  - 2011-05-03 release.r57 & release.r57.1
  - 2011-03-06 release.r56
  - … (only weeklies prior to r56)
  - 2009-11-06 weekly.2009-11-06
    - First release on GitHub

Go version 1 (Go 1) defines two things:
- the specification of the language
- the specification of the standard packages

It is intended that programs written to the Go 1 specification will continue to compile and run correctly, unchanged, over the lifetime of that specification

At some indefinite point, a Go 2 specification may arise, but until that time, Go programs that work today should continue to work even as future "point" releases of Go 1 arise

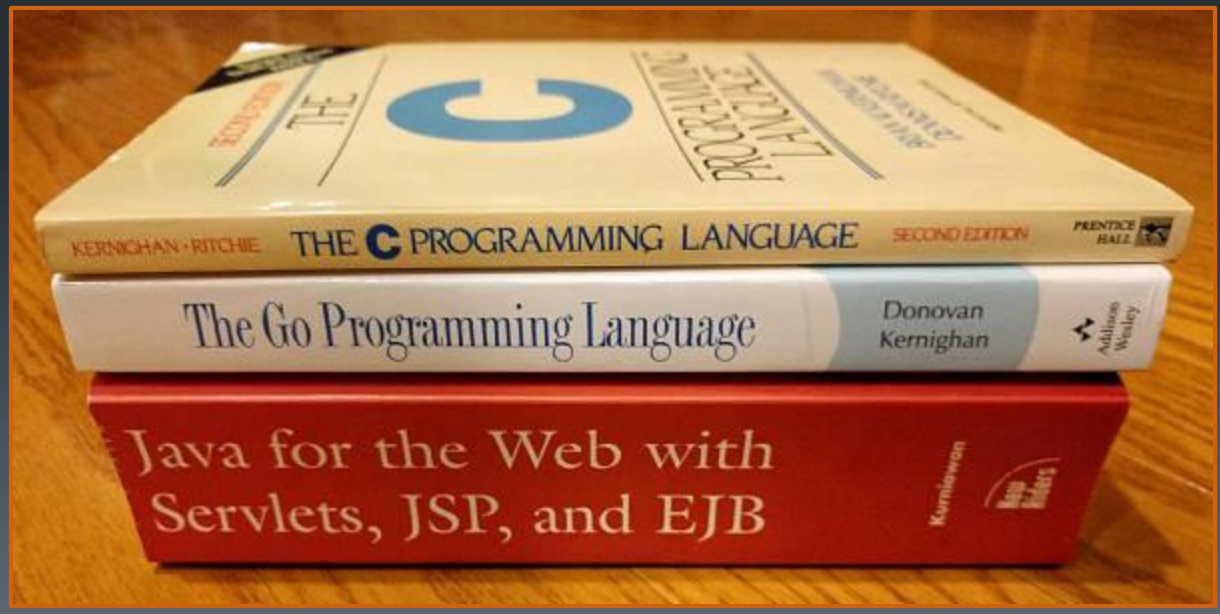
Compatibility is at the source level (binary compatibility for compiled packages is not guaranteed between releases)

APIs may grow, acquiring new packages and features, but not in a way that breaks existing Go 1 code

Go has been post v1.0 for over 5 years
And in use for over 10 years

Go is compact

- K&R C has 32 keywords
- C99 has 37 keywords
- C11 has 44 keywords
- Go has 25 keywords + 18 reserved type and constant names
- Java has 53 keywords
- The current version of C++ has 94 keywords not counting preprocessor directives



High level style comparison

- Go provides an unique mix of features
  - Simple syntax
  - Compiled language performance
  - Static linking runtime simplicity
  - Robust concurrency
  - Absence of OO inheritance
  - A unique approach to interfaces with implicit implementation
  - Memory safety
    - Which can be disabled for low level development
  - A robust set of modern libraries making web and network service development easy

Language comparison

	Python	Ruby	JS/ Node.js	C/C++	Java	Go
semicolons	N	N	Y	Y	Y	N
curly braces	N	N*	Y	Y	Y	Y
static types	N	N	N	Y	Y	Y
easy-to-use concurrency	N	N	Y	N	N	Y
multi-core concurrency	N	N	N	Y	Y	Y
compiled	N	N	N	Y	Y	Y
OO: classes, inheritance	Y	Y	Y	Y	Y	N*

Hello World in Go

- example.go
  - Go source files end with .go
- package main
  - All Go code must exist within a package
  - The first statement in a source file must declare its package
  - The “main” package generates an executable, all other package names generate a library package
- import “fmt”
  - You must import other packages you will use in your code
  - The fmt package is a standard library package that reads and writes formatted text
- func main() {
  - Go programs declare functions with the func keyword
  - The main() function is special when found in the main package and is used to launch an application
  - Function arguments are defined within parenthesis (our example main function receives no arguments)
  - The body of a function is enclosed in curly braces
- fmt.Println(“Hello world”)
  - fmt.Println() displays parameters to stdout

Go takes a strong stance on code formatting, declaring a standard format by fiat. This eliminates debate and enables a variety of automated source code transformations and tooling. The gofmt tool rewrites code into the standard format.

Packages

- Go code is organized into packages
  - Similar to libraries or modules in other languages
- A package consists of one or more .go source files in a single directory that define what the package does
- Each source file begins with a package declaration
- Followed by the necessary package imports
  - The Go standard library has over 100 packages
  - You must import exactly the packages you need
    - A program will not compile if there are missing imports or if there are unnecessary ones
- Package main defines an executable program as opposed to a library
- Within package main the function main is where execution of the program begins

```
user@ubuntu:~/go$ cat main.go
package main

import "fmt"

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

```
1  package consul
2
3  import (
4          "errors"
5          "fmt"
6          "os"
7          "testing"
8          "time"
9
10         "github.com/hashicorp/consul/consul/structs"
11         "github.com/hashicorp/consul/testutil"
12         "github.com/hashicorp/net-rpc-msgpackrpc"
13         "github.com/hashicorp/serf/serf"
14 )
15
```

go

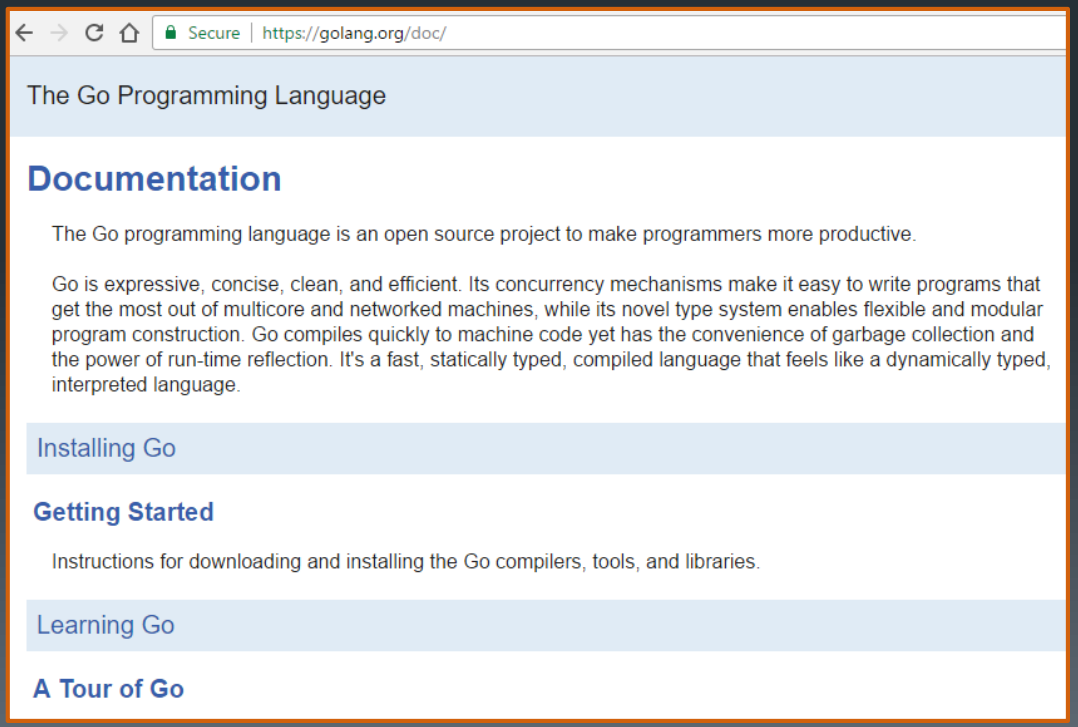
- The go command provides a unified front end to all of the key go tools
  - go run
    - Compiles and runs a program, then cleans the build
  - go build
    - Compiles programs
  - go clean
    - Removes intermediate files
  - go doc
    - Displays the documentation for a package or an exported package feature

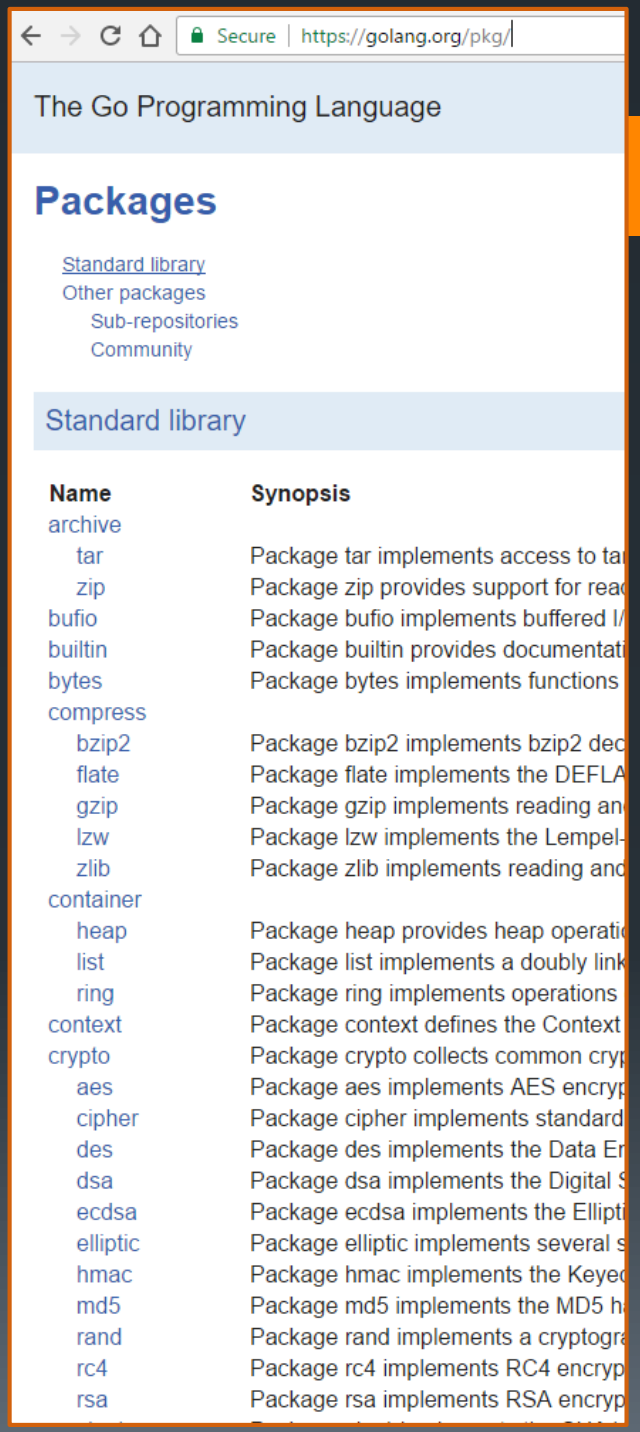
```
user@ubuntu:~$ go
Go is a tool for managing Go source code.

Usage:

        go command [arguments]

The commands are:

        build       compile packages and dependencies
        clean       remove object files
        doc         show documentation for package or symbol
        env         print Go environment information
        bug         start a bug report
        fix         run go tool fix on packages
        fmt         run gofmt on package sources
        generate    generate Go files by processing source
        get         download and install packages and dependencies
        install     compile and install packages and dependencies
        list        list packages
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
        vet         run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

        c           calling between Go and C
        buildmode   description of build modes
                                                    riable


                                                 e lists
                                                 g flags
                                                 g functions

                                                 about that topic.

user@ubuntu:~$
```

```
user@ubuntu:~/go/lab01$ 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.
```

Documentation

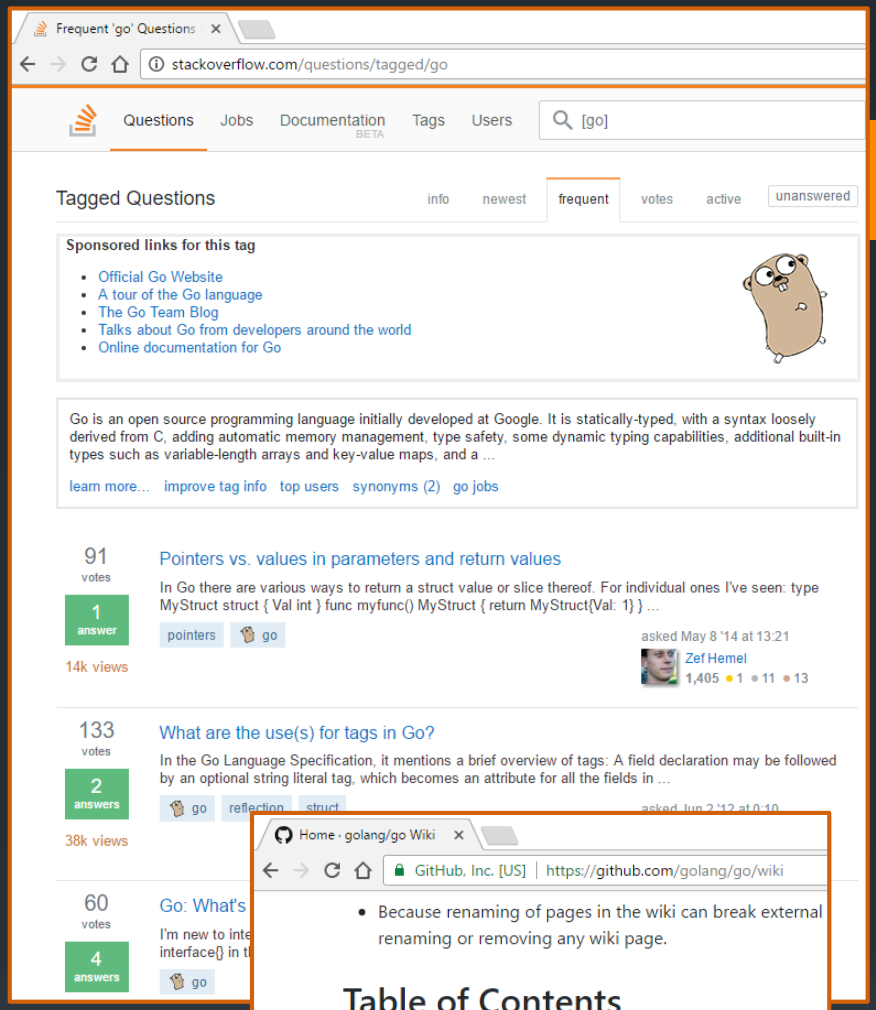
- General Documentation:
  - Getting started, tours, etc.
  - https://golang.org/doc/
- Package Reference:
  - The Go Standard Library contains a wide range of packages
    - > 100 packages
    - You will spend a lot of time here
  - https://golang.org/pkg/





Resources

- Stack Overflow – Go tag
  - Over 17k followers
  - Over 21k questions
- Go Nuts Mailing List - https://groups.google.com/group/golang-nuts
  - Get help from Go users, and share your work on the official mailing list.
  - Search the golang-nuts archives and consult the FAQ and wiki before posting.
- Go Forum - https://forum.golangbridge.org/
  - The Go Forum is a discussion forum for Go programmers.
- Gopher Slack - https://blog.gopheracademy.com/gophers-slack-community/
  - Get live support from other users in the Go slack channel.
- Go IRC Channel - #go-nuts on irc.freenode.net
  - Get live support at #go-nuts on irc.freenode.net, the official Go IRC channel.
- Frequently Asked Questions (FAQ) - https://golang.org/doc/faq
  - Answers to common questions about Go.
- Announcements Mailing List - https://groups.google.com/group/golang-announce
  - Stay informed on the Go announcements mailing list
  - Subscribe to golang-announce for important announcements, such as the availability of new Go releases.
- Go Blog - https://blog.golang.org/
  - The Go project's official blog.
- @golang at Twitter
  - The Go project's official Twitter account.
- Go+ community
  - A Google+ community for Go enthusiasts.
- golang sub-Reddit - https://reddit.com/r/golang
  - The golang sub-Reddit is a place for Go news and discussion.
- Go User Groups - https://golang.org/wiki/GoUserGroups
  - Each month in places around the world, groups of Go programmers ("gophers") meet to talk about Go. Find a chapter near you.
- Go Playground - https://golang.org/play
  - A place to write, run, and share Go code.
- Go Wiki - https://golang.org/wiki
  - A wiki maintained by the Go community.



Go Editors/IDEs

- One of the guiding directives for Go is:
  - No need for integrated development environments, but optional support for them
- A range of editors and IDEs are commonly used with Go

- VIM!
  - vim-go
- EMACS
  - You know the type …
- Sublime Text
  - sublime-build plugin for go
- Lime
  - A Sublime lookalike written in Go (!)
  - Not ready for prime time
- Atom
  - go-plus plugin

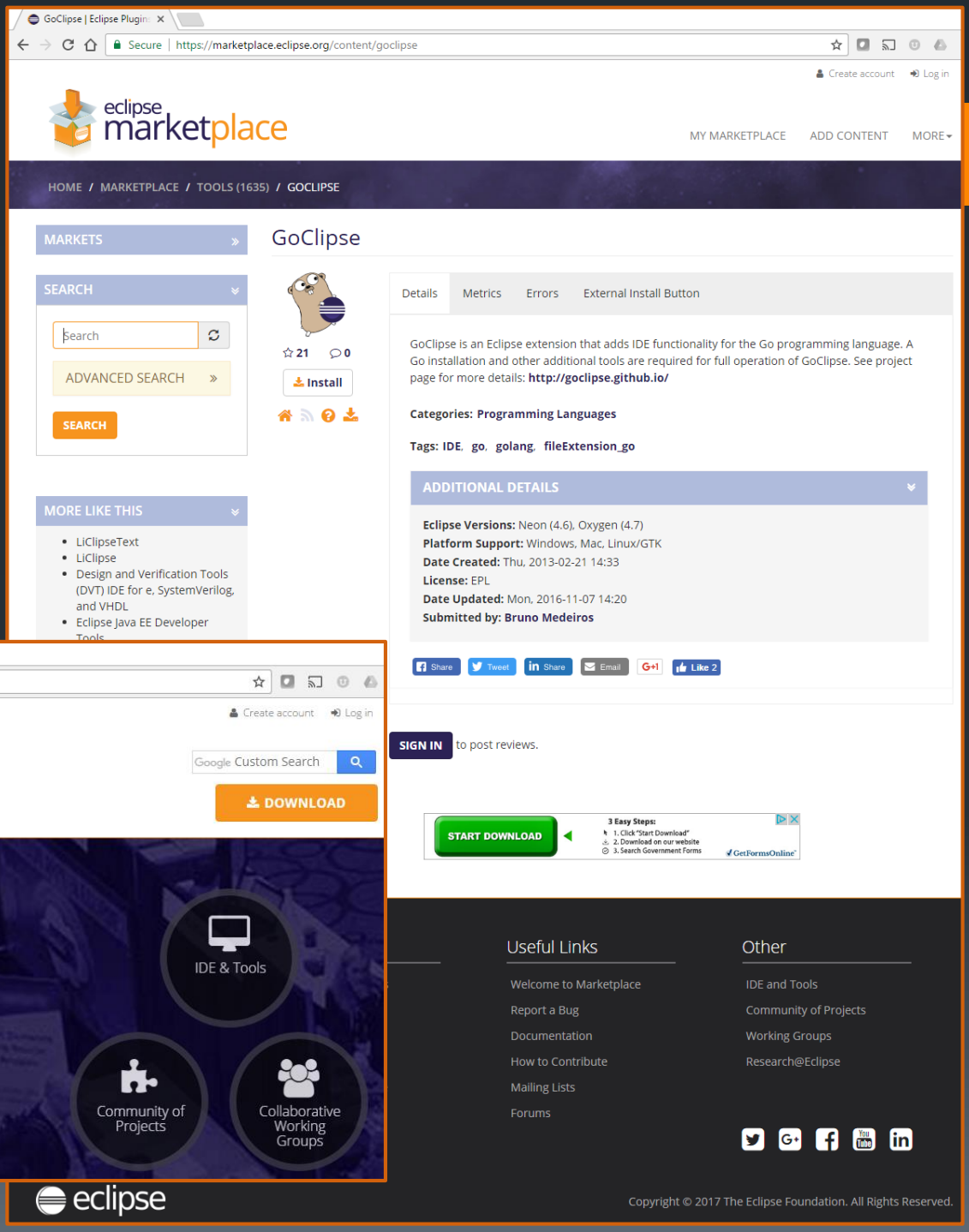
- IDEs with debuggers:
  - Eclipse
    - Goclipse
  - Visual Studio Code
    - vscode-go plugin
  - JetBrains Gogland

- Common Go editor features
  - code completion
  - automatic imports
  - automatic code formatting
  - linter integration
  - code navigation
- Uncommon features:
  - debugging (not found in straight editor plugins)

- Go tools driving features under the hood
  - gocode: completion lists
  - godoc: signature help
  - godef: Goto definition
  - guru: finding references
  - go-outline: file outlining
  - go-symbols: workspace symbol search
  - gorename: rename identifiers
  - go build and go test: build support
  - golint: linter
  - gometalinter: linter
  - gofmt and goimports and goreturns: formatting
  - gopkgs: import resolution
  - delve: debugger

Eclipse

- Eclipse is a popular cross platform IDE famous for its Java development environment
  - Eclipse is written in Java
- The GoClipse plugin adds support for Go development to Eclipse



Summary

- Go was created to define a language without the key problems of prevalent languages while preserving their strengths
- Go was designed to be simple to use yet highly productive
- Go includes a large standard library empowering important modern features such as web service implementation
- Go programs are divided into packages
- Go eco system has grown tremendously over the last decade
- Go documentation is robust and easy to navigate

Lab: Overview + Optional GoClipse

- Setting up Go tools and creating Go programs

2: Syntax

Objectives

- Learn basic syntax
- Understand Go variables, functions, conditionals, and loops
- Explain scope and lifetime in Go

Comments

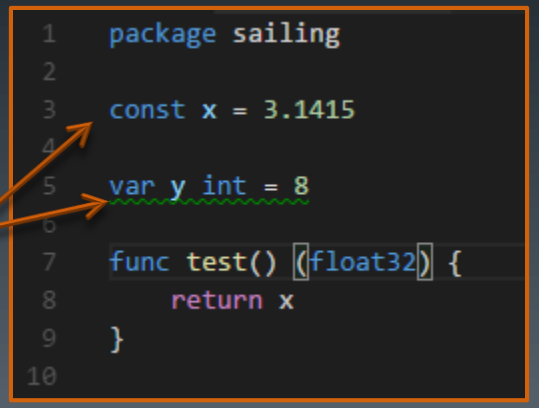
- Comments are ignored by the Go compiler
  - Used by humans to document and delineate
- Go supports two forms of comments
  - //
    - “Line Comments”
    - Begins a comment which terminates at the end of the line
    - Used in most cases
  - /* */
    - “Block comments”
    - Everything between the *s is part of the comment
    - Can include multiple lines
      - Often used to disable large blocks of code
    - Also useful within an expression
    - Used for package comments
      - Every package should have a package comment, a block comment preceding the package clause
      - The package comment should introduce the package and provide information relevant to the package as a whole

```
/*
Package regexp implements a simple library for regular expressions.

The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

```
163
164         // At this point, the initialization of etcd is done.
165         // The listeners are listening on the TCP ports and ready
166         // for accepting connections. The etcd instance should be
167         // joined with the cluster and ready to serve incoming
168         // connections.
169         notifySystemd()
170
171         select {
172         case lerr := <-errc:
173                 // fatal out on listener errors
174                 plog.Fatal(lerr)
175         case <-stopped:
176         }
177
178         osutil.Exit(0)
179 }
180
181 // startEtcd runs StartEtcd in addition to hooks needed for standalone etcd.
182 func startEtcd(cfg *embed.Config) (<-chan struct{}, <-chan error, error) {
183         if cfg.Metrics == "extensive" {
184                 grpc_prometheus.EnableHandlingTimeHistogram()
185         }
186
```

Declarations

- A Go program consists principally of 5 types of declarations:
  - Packages – package
  - Functions – func
  - Variables – var
  - Constants – const
  - Types – type
- Declarations begin with the keyword that specifies the declaration type, are followed by the declaration identifier and end with the specification of the thing
  - var x int = 5
    - Type can be inferred on initialization
  - func x …
  - const x int = 6
    - When type can be inferred idiomatic Go leaves it out
  - package sailing …



Functions

- A function declaration consists of:
  - The keyword func
  - The name of the function
  - A parameter list (can be empty)
  - A result list (can be empty)
  - The body of the function
- The statements that define a function's behavior are enclosed in braces
- Execution of the function begins with the first statement and continues until it encounters a return statement or reaches the end of a function that has no results
  - Control and any results are then returned to the caller
- Go requires no semicolons
- Go supports first class functions, higher-order functions, user-defined function types, function literals, closures, and multiple return values

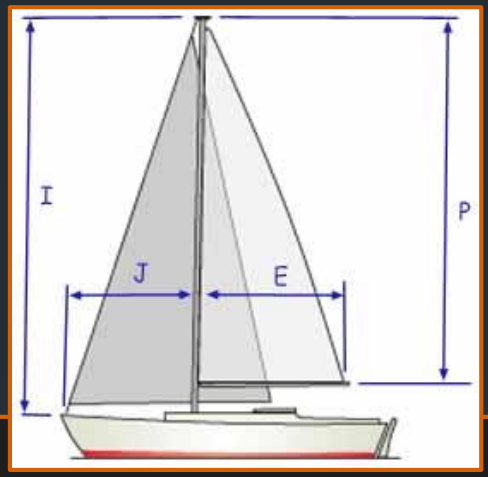
```
67  // ensurePath is used to make sure a path exists
68  func ensurePath(path string, dir bool) error {
69          if !dir {
70                  path = filepath.Dir(path)
71          }
72          return os.MkdirAll(path, 0755)
73  }
```

```
104 // Returns if a member is a consul node. Returns a bool,
105 // and the datacenter.
106 func isConsulNode(m serf.Member) (bool, string) {
107         if m.Tags["role"] != "node" {
108                 return false, ""
109         }
110         return true, m.Tags["dc"]
111 }
```

```
268 // runtimeStats is used to return various runtime information
269 func runtimeStats() map[string]string {
270         return map[string]string{
271                 "os":         runtime.GOOS,
272                 "arch":       runtime.GOARCH,
273                 "version":    runtime.Version(),
274                 "max_procs":  strconv.FormatInt(int64(runtime.GOMAXPROCS(0)), 10),
275                 "goroutines": strconv.FormatInt(int64(runtime.NumGoroutine()), 10),
276                 "cpu_count":  strconv.FormatInt(int64(runtime.NumCPU()), 10),
277         }
278 }
```

Variables

- var declarations create a variable with a
  - Type
  - Identifier
  - Initial value
- General form: `var name type = expression`
  - Either type or the expression may be omitted
  - If the expression is omitted the identifier is initialized to the "zero value" for that type
    - 0 for numbers
    - False for Booleans
    - "" for strings
    - nil for nilable types
  - Go does not allow uninitialized variables
  - Suggests making the zero value a meaningful marker within your app




```go
package main

import "sailing"
import "fmt"

func main() {
    var i float32 = 77.5
    var j float32 = 23.3
    var e float32 = 21.7
    var p float32 = 74.3
    fmt.Println("Main area", sailing.CalcM(e, p))
    fmt.Println("Foretriangle", sailing.CalcFT(j, i))
    fmt.Println("Sail Area", sailing.CalcSailArea(e, p, j, i))
}
```

```go
package sailing

//CalcM returns the main sail area
func CalcM(e float32, p float32) float32 {
    return e * p / 2
}

//CalcFT returns the fore triangle (jib sail area)
func CalcFT(j float32, i float32) float32 {
    return j * i / 2
}

//CalcSailArea returns the total sail area
func CalcSailArea(e float32, p float32, j float32, i float32) float32 {
    return CalcM(e, p) + CalcFT(j, i)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
API server listening at: 127.0.0.1:29228
Main area 806.1551
Foretriangle 902.875
Sail Area 1709.03
```

Expressions

- Assignment
  - x = 9
  - Each arithmetic and bitwise binary operator has a corresponding combined assignment operator
    - x += 7, x *=2
  - Tuple assignment allows several variables to be assigned at once by evaluating all of the right side variables before any variables are updated
    - x, y = y, x
  - The blank identifier “_” can absorb unwanted assigned values
    - x, _ = y, x
    - _, y = ftComponents()
- Increment/decrement
  - x++, x--
  - No prefix inc/dec

Binary Operators

- 5 Levels of precedence (from high to low):
  - * / % << >> & &^
  - + - | ^
  - == != < <= > >=
  - &&
  - ||
- Operators in each category are left associative
  - Parenthesis can be used to force ordering
  - mask & (1 < < 28)
- The sign of the remainder produced by % is always the same as the sign of the dividend (the number to the left of the operator)
- Integer division truncates the result toward 0
- Overflows truncate high order bits

+ and – also have unary forms
(+ has no effect and – negates the value it precedes)

& | ^ &^ << >> are bitwise
(AND, OR, XOR, AND NOT, Left Shift and Right Shift respectively)
<< fill 0, >> of unsigneds fill 0, >> of signeds fill the sign bit
^ is bitwise NOT in unary form

example.go

```go
package main

import "fmt"

func main() {
    var x uint8 = 1<<1 | 1<<5
    var y uint8 = 1<<1 | 1<<2
    fmt.Printf("%08b\n", x)
    fmt.Printf("%08b\n", y)
    fmt.Printf("%08b\n", x&y)
    fmt.Printf("%08b\n", x|y)
    fmt.Printf("%08b\n", x^y)
    fmt.Printf("%08b\n", x&^y)
    for i := uint(0); i < 8; i++ {
        if x&(1<<i) != 0 {
            fmt.Println(i)
        }
    }
    fmt.Printf("%08b\n", x<<1)
    fmt.Printf("%08b\n", x>>1)
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
API server listening at: 127.0.0.1:39274
00100010
00000110
00000010
00100110
00100100
00100000
1
5
01000100
00010001
```

Compact initializers

- Multiple variables can be initialized with a single var statement
  - If the type is omitted it is inferred from the initializer
  - This allows multiple variables of different types to be initialized on a single line
  - Multiple variables can be initialized with a function that returns the correct number of values
  - Multiple variables can also be initialized in a var block within parenthesis
- The := operator can be used to initialize variables within function definitions
  - When it can be used, this is the preferred syntax
  - If some of the left side variables already exists they are assigned to rather than created
    - At least one of the variables must be created in the expression for it to compile (otherwise assignment "=" should be used)
- The default type for floats is float64
  - You can cast these to float using the Go "type()" cast syntax
  - The var keyword is often used when an explicit non-default type is required of when there is no explicit initial value for the identifier

```go
package main

import "fmt"

func main() {
    const (
        x = 3
    )
    var (
        t = 342
        u = 42
    )
    fmt.Println(u, t, x)
}
```

```go
package main

import (
    "fmt"
    "sailing"
)

func ftm(e float32, p float32, j float32, i float32) (float32, float32) {
    return sailing.CalcFT(j, i), sailing.CalcM(e, p)
}

func main() {
    var i, j, e, p float32 = 77.5, 23.3, 21.7, 74.3
    var ft, m = sailing.CalcFT(j, i), sailing.CalcM(e, p)
    fmt.Println("FT, M", ft, m)
    var ft2, m2 = ftm(e, p, j, i)
    fmt.Println("FT, M", ft2, m2)
    fmt.Println("Main area", sailing.CalcM(e, p))
    fmt.Println("Foretriangle", sailing.CalcFT(j, i))
    fmt.Println("Sail Area", sailing.CalcSailArea(e, p, j, i))

    j2 := 24.1
    e2, p2 := 22.4, 75.9
    fmt.Println("Alt Foretriangle", sailing.CalcFT(float32(j2), i))
    fmt.Println("Alt Main", sailing.CalcFT(float32(e2), float32(p2)))
}
```

Lifetime

- The lifetime of a variable is the interval of time during which it exists as the program executes
- Package-level variables live for the entire execution of the program
- Local variables have dynamic lifetimes:
  - Live until they become unreachable
    - At which point its storage may be recycled
  - A new instance is created each time the declaration statement is executed
  - Function parameters and results are local variables
- The compiler (not the programmer) decides whether to allocate a variable on the stack or the heap
  - Locals referenced outside of their function are said to have “escaped” their function/loop and must be heap allocated

Conditional and Switch Statements

- Similar to C syntax
  - Don't require parenthesis
  - Do require curly braces
  - An if statement can appear without an else statement
  - A variable can be created on the fly within an if statement
  - There is no ternary if in Go
    - e.g. a ? b : c

```
package main

import "fmt"

func main() {
    if 4 % 2 == 0 {
        fmt.Println("4 is even")
    } else {
        fmt.Println("4 is odd")
    }

    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "is a one digit number")
    } else {
        fmt.Println(num, "has multiple digits")
    }

}
```

```
i := 2
switch i {
case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two")
case 3:
    fmt.Println("three")
}
```

- A basic switch statement
  - Can also use 'default' as an alternate way to express if/else logic
  - Can use commas to separate multiple values in one case statement

Loops

- There is no ‘while’ loop in Go
- Compact initialization can (and should) be used
- Break and continue have the same usage as they do in Python

```go
// `for` is Go's only looping construct. Here are
// three basic types of `for` loops.

package main

import "fmt"

func main() {

    // The most basic type, with a single condition.
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // A classic initial/condition/after `for` loop.
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // `for` without a condition will loop repeatedly
    // until you `break` out of the loop or `return` from
    // the enclosing function.
    for {
        fmt.Println("loop")
        break
    }

    // You can also `continue` to the next iteration of
    // the loop.
    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

Source: https://gobyexample.com/for

Summary

- Go syntax
- Go variables
- Go variable scope and lifetime

Lab: Syntax and Flow Control

- Gain familiarity with writing Go code

3: Data Types and Composite Types

Objectives

- Gain an introduction to type system
- Understand:
  - Integers
  - Floats
  - Strings
  - Key type manipulation packages
  - Arrays
  - Slices
  - Maps

Go Types

- Go's types fall into four categories:
  - basic types
    - numbers, strings, and booleans
  - aggregate types
    - arrays and structs
  - reference types
    - pointers, slices, maps, functions, and channels
  - interface types
    - a way to specify the behavior of an object

Integers

Printed with fmt functions using:
%d %o %x and %b
decimal, octal, hex and binary respectively

- Go supports several sizes of signed and unsigned integers
- There are four distinct sizes of signed/unsigned integers
  - int8/uint8 - 8 bits
  - int16/uint16 - 16 bits
  - int32/uint32 - 32 bits
  - int64/uint64 - 64 bits
  - Unsigneds are avoided and used only when bitwise operations are required on binary data
- There are also (most efficient size for signed and unsigned integers on a particular platform)
  - int
  - uint
  - Both have the same size (either 32 or 64 bits)
  - Different compilers may make different choices even on identical hardware
- rune is a synonym for int32
  - Indicates that a value is a Unicode code point
  - The two names may be used interchangeably
- byte is a synonym for uint8
  - emphasizes that the value is a piece of raw data rather than a small numeric quantity
- uintptr has unspecified width but is sufficient to hold all the bits of a pointer value
  - used only for low-level programming (at the boundary of a Go program and a C library or OS)
- int, uint, and uintptr are different types from their explicitly sized siblings

Floating Point

- Go provides two sizes of floating-point numbers
  - float32
  - float64
- IEEE 754 standard implemented by all modern CPUs
- The math package provides float limits
  - math.MaxFloat32 (about 3.4e38)
  - math.MaxFloat64 (about 1.8e308)
  - math.MinFloat32 (about 1.4e-45)
  - math.MinFloat64 (about 4.9e-324)
- Precision
  - float32 provides approximately 6 decimal digits of precision
  - float64 provides approximately 15 decimal digits of precision
- float64 should be preferred to avoid cumulative errors
- Scientific notation is supported
  - const Avogadro = 6.02214129e23
  - const Planck = 6.62606957e-34
- math.NaN() equates to the result operations such as 0/0 or math.Sqrt(-1)
  - %f will display “NaN”

Printed with fmt functions using:
%g, %e, %f
General, exponent and normal respectively

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    for x := 0; x < 8; x++ {
        fmt.Printf("x = %d ex = %8.3f\n", x, math.Exp(float64(x)))
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
API server listening at: 127.0.0.1:14975
x = 0 ex =    1.000
x = 1 ex =    2.718
x = 2 ex =    7.389
x = 3 ex =   20.086
x = 4 ex =   54.598
x = 5 ex =  148.413
x = 6 ex =  403.429
x = 7 ex = 1096.633
```

Complex Numbers

- Go provides two sizes of complex numbers
  - complex64
  - complex128
- These have real and imaginary components which are float32 and float64 respectively
- The built-in function complex creates a complex number from its real and imaginary components
  - var x complex128 = complex(1, 2)
  - Also using the “i” suffix
    - x := 1 + 2i
    - y := 3 + 4i
- The built-in real and imag functions extract those components
  - fmt.Println(real(x*y))
  - fmt.Println(imag(x*y))
- The standard math.cmplx package contains math functions for complex numbers

```
user@ubuntu:~$ go doc math/cmplx
package cmplx // import "math/cmplx"

Package cmplx provides basic constants and mathematical functions for
complex numbers.

func Abs(x complex128) float64
func Acos(x complex128) complex128
func Acosh(x complex128) complex128
func Asin(x complex128) complex128
func Asinh(x complex128) complex128
func Atan(x complex128) complex128
func Atanh(x complex128) complex128
func Conj(x complex128) complex128
func Cos(x complex128) complex128
func Cosh(x complex128) complex128
func Cot(x complex128) complex128
func Exp(x complex128) complex128
func Inf() complex128
func IsInf(x complex128) bool
func IsNaN(x complex128) bool
func Log(x complex128) complex128
func Log10(x complex128) complex128
func NaN() complex128
func Phase(x complex128) float64
func Polar(x complex128) (r, θ float64)
func Pow(x, y complex128) complex128
func Rect(r, θ float64) complex128
func Sin(x complex128) complex128
func Sinh(x complex128) complex128
func Sqrt(x complex128) complex128
func Tan(x complex128) complex128
func Tanh(x complex128) complex128
user@ubuntu:~$
```

Booleans

- Booleans are declared:
  - bool
- Two possible values:
  - true
  - false
- There is no implicit conversion of a bool to a numeric
- Conditions in if and for statements are Booleans
- Comparison operators like == and < produce Booleans
- The unary operator ! is logical negation
  - “x == true” can be simplified to “x” (“x” being preferred)
- Boolean values can be combined && (AND) and | | (OR)
  - && has precedence
  - short-circuit behavior

```
# _/d_/dev/go/example/src
.\example.go:16: cannot use x (type bool) as type int in assignment
exit status 2
Process exiting with code: 1
```

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       var x bool = true
9       var y bool = false
10      fmt.Println(x, y)
11      fmt.Println(x || y)
12      fmt.Println(x && y)
13      fmt.Println(!x)
14  }
15
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
API server listening at: 127.0.0.1:49175
true false
true
false
false
```

Strings

- A string is an immutable sequence of bytes
  - string
- Strings may contain arbitrary data, including bytes with value 0, but usually they contain human-readable text
- The len function returns the number of bytes (not runes) in a string
  - The i-th byte of a string is not necessarily the i-th character of a string because the UTF-8 encoding of a non-ASCII code point requires two or more bytes
- The index operation s[i] retrieves the i-th byte of string s
  - where 0 ≤ i < len(s)
  - Attempting to access a byte outside this range results in a panic
  - s[3] = 'x' //illegal, strings are immutable
  - s = "x" //ok, s gets new string (old string unch)
- The substring operation s[i:j] yields a new string consisting of the bytes of the original string starting at index i and continuing up to, but not including, the byte at index j
  - Omitting the i starts the substring at 0
  - Omitting the j stops the substring at len(s)
- Substrings share memory of the original (immutable) string
- Strings may be:
  - + Concatenated
  - += Concatenated and assigned back to the var
  - ==, <, <=, >, >=, != Compared lexically (byte by byte)
- String literals:
  - "Hello world\n" //Hello world with a new line
  - `Hello world\n` //Hello world\n (raw)
    - Good for json, html, regex, etc.

\a	"alert" or bell
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\'	single quote (only in the rune literal '\'')
\"	double quote (only within "..." literals)
\\	backslash

escape sequences

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      str := "Now is the time for all good men to " +
7          "come to the aid of their country"
8      fmt.Println(str)
9      fmt.Println(len(str))
10     fmt.Println(str[len(str)-1])
11     fmt.Printf("%c\n", str[len(str)-1])
12     fmt.Println(str[:15])
13     fmt.Println(str[11:15])
14     fmt.Println(str[11:])
15     b := []byte(str)
16     fmt.Println(b)
17 }
18
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
API server listening at: 127.0.0.1:7505
Now is the time for all good men to come to the aid of their country
68
121
y
Now is the time
time
time for all good men to come to the aid of their country
[78 111 119 32 105 115 32 116 104 101 32 116 105 109 101 32 102 111 1
00 32 111 102 32 116 104 101 105 114 32 99 111 117 110 116 114 121]
```

String libs

- Four standard packages are important for manipulating strings:
  - bytes
    - functions for manipulating slices of type [] byte
    - strings are immutable and building strings incrementally can be expensive
    - it's more efficient to use the bytes.Buffer
  - strings
    - functions for searching, replacing, comparing, trimming, splitting, and joining strings
  - strconv
    - functions for converting boolean, integer, and floating-point values to and from their string representations
    - functions for quoting and unquoting strings
  - unicode
    - functions like IsDigit, IsLetter, IsUpper, and IsLower for classifying runes

```
user@ubuntu:~$ go doc strings
package strings // import "strings"

Package strings implements simple functions to manipulate UTF-8 encoded
strings.

For information about UTF-8 strings in Go, see
https://blog.golang.org/strings.

func Compare(a, b string) int
func Contains(s, substr string) bool
func ContainsAny(s, chars string) bool
func ContainsRune(s string, r rune) bool
func Count(s, sep string) int
func EqualFold(s, t string) bool
func Fields(s string) []string
func FieldsFunc(s string, f func(rune) bool) []string
func HasPrefix(s, prefix string) bool
func HasSuffix(s, suffix string) bool
func Index(s, sep string) int
func IndexAny(s, chars string) int
func IndexByte(s string, c byte) int
func IndexFunc(s string, f func(rune) bool) int
func IndexRune(s string, r rune) int
func Join(a []string, sep string) string
func LastIndex(s, sep string) int
func LastIndexAny(s, chars string) int
func LastIndexByte(s string, c byte) int
func LastIndexFunc(s string, f func(rune) bool) int
func Map(mapping func(rune) rune, s string) string
func Repeat(s string, count int) string
func Replace(s, old, new string, n int) string
func Split(s, sep string) []string
func SplitAfter(s, sep string) []string
func SplitAfterN(s, sep string, n int) []string
func SplitN(s, sep string, n int) []string
func Title(s string) string
func ToLower(s string) string
func ToLowerSpecial(c unicode.SpecialCase, s string) string
func ToTitle(s string) string
func ToTitleSpecial(c unicode.SpecialCase, s string) string
func ToUpper(s string) string
func ToUpperSpecial(c unicode.SpecialCase, s string) string
func Trim(s string, cutset string) string
func TrimFunc(s string, f func(rune) bool) string
func TrimLeft(s string, cutset string) string
func TrimLeftFunc(s string, f func(rune) bool) string
func TrimPrefix(s, prefix string) string
func TrimRight(s string, cutset string) string
func TrimRightFunc(s string, f func(rune) bool) string
func TrimSpace(s string) string
func TrimSuffix(s, suffix string) string
type Reader struct{ ... }
    func NewReader(s string) *Reader
type Replacer struct{ ... }
    func NewReplacer(oldnew ...string) *Replacer

BUG: The rule Title uses for word boundaries does not handle Unicode punctuation properly.

user@ubuntu:~$
```

Unicode

- Text strings are conventionally interpreted as UTF-8-encoded sequences of Unicode code points (runes)
  - UTF-8 was invented by Ken Thompson and Rob Pike, two of the creators of Go
- Go source files are always in UTF-8

```
user@ubuntu:~$ go doc unicode
package unicode // import "unicode"

Package unicode provides data and functions to test some properties of
Unicode code points.

const MaxRune = '\U0010FFFF' ...
const UpperCase = iota ...
const UpperLower = MaxRune + 1
const Version = "9.0.0"
var Cc = _Cc ...
var Adlam = _Adlam ...
var ASCII_Hex_Digit = _ASCII_Hex_Digit ...
var CaseRanges = _CaseRanges
var Categories = map[string]*RangeTable{ ... }
var FoldCategory = map[string]*RangeTable{ ... }
var FoldScript = map[string]*RangeTable{}
var GraphicRanges = []*RangeTable{ ... }
var PrintRanges = []*RangeTable{ ... }
var Properties = map[string]*RangeTable{ ... }
var Scripts = map[string]*RangeTable{ ... }
func In(r rune, ranges ...*RangeTable) bool
func Is(rangeTab *RangeTable, r rune) bool
func IsControl(r rune) bool
func IsDigit(r rune) bool
func IsGraphic(r rune) bool
```

```
1  package main
2
3  import "fmt"
4  import "unicode/utf8"
5
6  func main() {
7      str := "Hello \xe4\xb8\x96\xe7\x95\x8c"
8      fmt.Println(str)
9      fmt.Println(len(str))
10     fmt.Println(utf8.RuneCountInString(str))
11 }
12
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
API server listening at: 127.0.0.1:27339
Hello 世界
12
8
```

```
user@ubuntu:~$ go doc unicode/utf8
package utf8 // import "unicode/utf8"

Package utf8 implements functions and constants to support text encoded in
UTF-8. It includes functions to translate between runes and UTF-8 byte
sequences.

const RuneError = '\uFFFD' ...
func DecodeLastRune(p []byte) (r rune, size int)
func DecodeLastRuneInString(s string) (r rune, size int)
func DecodeRune(p []byte) (r rune, size int)
func DecodeRuneInString(s string) (r rune, size int)
func EncodeRune(p []byte, r rune) int
func FullRune(p []byte) bool
func FullRuneInString(s string) bool
func RuneCount(p []byte) int
func RuneCountInString(s string) (n int)
func RuneLen(r rune) int
func RuneStart(b byte) bool
func Valid(p []byte) bool
func ValidRune(r rune) bool
func ValidString(s string) bool
user@ubuntu:~$
```

```
    var AzeriCase SpecialCase = _TurkishCase
    var TurkishCase SpecialCase = _TurkishCase

BUG: There is no mechanism for full case folding, that is, for
characters that involve multiple runes in the input or output.

user@ubuntu:~$
```

Arrays

- An array is a fixed-length sequence of zero or more elements of a particular type
- Because of their fixed length, arrays are rarely used directly in Go
  - Slices, which can grow and shrink, are much more versatile
- Individual array elements are accessed with the conventional subscript notation x[i]
  - Subscripts run from zero to one less than the array length
- The built-in function len returns the number of elements in the array
- Array literals are enclosed in curly braces
  - 2 elements: {2, 4}
  - Sparse init: {3: 1.1, 5: 1.2}
- Ellipsis can be used to infer array size […]
  - The %T conversion displays an object's type

```go
package main

import "fmt"

func main() {
    var a [3]int
    fmt.Println(a[0])
    fmt.Println(a[len(a)-1])
    for i, v := range a {
        fmt.Printf("%d %d\n", i, v)
    }
    for _, v := range a {
        fmt.Printf("%d\n", v)
    }
    q := [...]int{1, 2, 3}
    fmt.Println(q)
    fmt.Printf("%T\n", q)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
API server listening at: 127.0.0.1:16837
0
0
0 0
1 0
2 0
0
0
0
[1 2 3]
[3]int
```

```go
package main

import "fmt"

func main() {
    r := [...]int{4: 42, 8: 42}
    for i2, v2 := range r {
        fmt.Printf("%d %d\n", i2, v2)
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
API server listening at: 127.0.0.1:18668
0 0
1 0
2 0
3 0
4 42
5 0
6 0
7 0
8 42
```

Slices

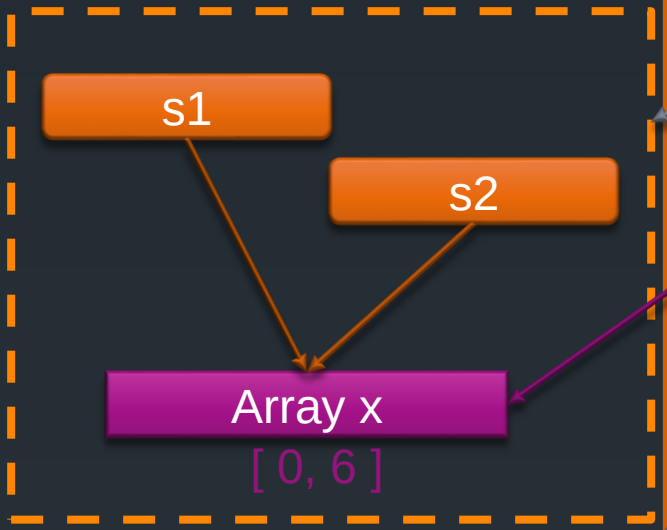
- A slice is a lightweight data structure that gives access to a subsequence (or perhaps all) of the elements of an array
  - known as the slice's underlying array
- Slices represent variable-length sequences whose elements all have the same type
- A slice type is written []T, where the elements have type T
  - Looks like an array type without a size
- A slice has three components:
  - A pointer
    - To the first element of the array reachable through the slice
  - A length
    - The number of slice elements (can't exceed the capacity)
    - Returned by the built-in function len()
  - A capacity
    - The number of elements between the start of the slice and the end of the underlying array
    - Returned by the built-in function cap()
- The built-in append function appends items to slices
  - summer = append(summer, "Freemonth")
  - Modifies underlying array!
- Unlike arrays, slices are not comparable
  - Cannot use == to test two slices for the same elements
- The slice zero value is nil

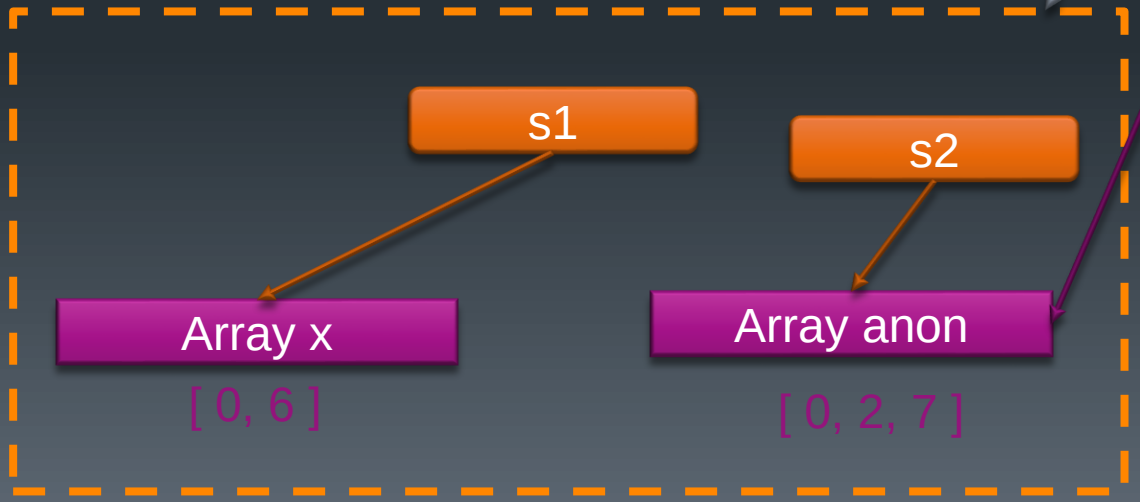
```go
package main

import (
    "fmt"
)

func main() {
    months := [...]string{1: "January", 2: "Febuary", 3: "March", 4: "April",
        5: "May", 6: "June", 7: "July", 8: "August",
        9: "September", 10: "October", 11: "November", 12: "December"}
    summer := months[6:9]
    fmt.Println(summer)
    fmt.Println(summer[1])
    fmt.Println(summer[:6])
    fmt.Println(summer)
    fmt.Println(len(summer))
    fmt.Println(cap(summer))
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/09 19:44:17 server.go:73: Using API v1
2017/04/09 19:44:17 debugger.go:68: launching process with args: [d:\dev\go\example\s
API server listening at: 127.0.0.1:2345
2017/04/09 19:44:17 debugger.go:414: continuing
[June July August]
July
[June July August September October November]
[June July August]
3
7
```

Append Internals

- Appending to a slice:
  - Increases the slice's length
  - If (and only if) the length exceeds the capacity of the underlying array the underlying array is copied to a new anonymous array with the necessary length and the slice capacity is updated to the same value as the length






```go
package main

import "fmt"

func main() {
    var x [2]int
    s1 := x[0:2]
    s2 := x[0:2]
    fmt.Println(x[1])
    fmt.Println(s1[1])
    fmt.Println(s2[1])
    s1[1] = 6
    fmt.Println(x[1])
    fmt.Println(s1[1])
    fmt.Println(s2[1])
    s1 = append(s1, 7)
    fmt.Println(x[1])
    fmt.Println(s1[1])
    fmt.Println(s2[1])
    s1[1] = 2
    fmt.Println(x[1])
    fmt.Println(s1[1])
    fmt.Println(s2[1])
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/08/13 14:27:58 server.go:73: Using API v1
2017/08/13 14:27:58 debugger.go:97: launching pro
API server listening at: 127.0.0.1:2345
2017/08/13 14:27:59 debugger.go:505: continuing
0
0
0
6
6
6
6
6
6
6
6
2
6
```

Maps

- A map is an unordered collection of key/value pairs
  - All keys are distinct
  - Value associated with a given key can be retrieved, updated, or removed
  - All operations are constant time O(1)
- Map types are written map[K] V
  - K is the key type
    - The key type K must be comparable using ==
    - Floating-point keys are almost always a bad idea as they are hard to compare for equality
  - V is the value type
- The built-in make function make can be used to create a map
  - motorbike := make(map[int32] string)
- Map literals are supported
  - motorbike := map[int32] string {
    1: “Honda”,
    2: “BMW”,
    }
- Maps are accessed through the hash key
  - var bike string = motorbike[1]
- The built in delete function removes elements
  - delete(motorbike, 1)
- Unknown keys return the zero value for the stored type in the above operations
- You can not take the address of a map element
- You can iterate over a map with a range based for loop
  - The order is however unspecified

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       x := map[int32]string{
7           1: "Honda",
8           2: "BMW",
9           3: "Ducati",
10          4: "Triumph",
11          5: "Polaris",
12      }
13      for k, v := range x {
14          fmt.Println(k, v)
15      }
16  }
17
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/09 22:39:45 server.go:73: Using API v1
2017/04/09 22:39:45 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/09 22:39:46 debugger.go:414: continuing
2 BMW
3 Ducati
4 Triumph
5 Polaris
1 Honda
```

Sets

- Go does not provide a set type
- However a map's keys are a set
- Defining a set of strings with a small ignored value type is common in go:
  - myset := make(map [string] bool)

Pointers

- A pointer value is the address of a variable
  - var x int
  - &x (address of x) yields a pointer to the integer variable
  - *int is type “pointer to int”
  - *p yields the value of the pointed to variable
- Every variable has an address
- The zero value for pointers is nil
- Pointers are equal only if they are both nil or they both point to the same variable
- In Go returning a pointer to a local variable is safe
  - Stop hyperventilating C++ programmers
  - Go maintains the memory as long as there are references to it

```
1   package main
2
3   import "fmt"
4
5   func dbl(d *float64) {
6       *d = *d + *d
7   }
8
9   func main() {
10      d := 23.9
11      dbl(&d)
12      fmt.Println(d)
13  }
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
API server listening at: 127.0.0.1:15969
47.8
```

New

- The built in new(type) function creates a new unnamed pointer variable and initializes it to the zero value for the specified type
- New is a built in function and can be masked by locals using the name “new”
  - It should be clear that this is a fairly bad idea

```go
package main

import "fmt"

func dbl(d *float64) {
    *d = *d + *d
}

func main() {
    a := new(float64)
    *a = 2
    dbl(a)
    fmt.Println(*a)
    fmt.Println(a)
    fmt.Println(&a)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
API server listening at: 127.0.0.1:41093
4
0xc0420441d0
0xc04205c018
```

Type Declarations

- A type declaration defines a new named type that has the same underlying type as an existing type
- Typically used at the package level for internal package types
- For every type T, there is a corresponding conversion operation T(x) that converts the value x to type T
  - A conversion from one type to another is allowed if both have the same underlying type

```
1   package main
2
3   import "fmt"
4
5   func test(x float64) float64 {
6       return x
7   }
8
9   func main() {
10      type USD float64
11      var bal USD = 45.7
12      test(bal)
13      fmt.Println(bal)
14  }
15
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
# _/d_/dev/go/example/src
.\example.go:12: cannot use bal (type USD) as type float64 in argument to test
exit status 2
Process exiting with code: 1
```

```
1   package main
2
3   import "fmt"
4
5   func test(x float64) float64 {
6       return x
7   }
8
9   func main() {
10      type USD float64
11      var bal USD = 45.7
12      test(float64(bal))
13      fmt.Println(bal)
14  }
15
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
API server listening at: 127.0.0.1:40719
45.7
```

Summary

- Concepts covered:
  - Integers
  - Floats
  - Strings
  - Key type manipulation packages
  - Slices
  - Arrays
  - Maps

Lab: Data Types and Composite Types

- Working with Types

4: Program Construction

Objectives

- Define Go program structure
- List the Go identifier rules
- Describe the function of packages and go get

Program Structure

- A Go program is stored in one or more files with the ".go" extension
- Files begin with a package declaration
  - This identifies the package the file is a part of
  - The folder name containing the files must match the package name
- The package declaration is followed by required import declarations
- Imports are followed by a sequence of package-level declarations of types, variables, constants, and functions, in any order

example.go

```
1   package main
2
3   import (
4       "fmt"
5       "sailing"
6   )
7
8   func main() {
9       var i float32 = 77.5
10      var j float32 = 23.3
11      var e float32 = 21.7
12      var p float32 = 74.3
13      fmt.Println("Main area", sailing.CalcM(e, p))
14      fmt.Println("Foretriangle", sailing.CalcFT(j, i))
15      fmt.Println("Sail Area", sailing.CalcSailArea(e, p, j, i))
16  }
17
```

```
Randy@romolack MINGW64 /d/dev/go/example
$ ls -l src/example.go
-rw-r--r-- 1 Randy 197609 322 Apr  8 21:53 src/example.go

Randy@romolack MINGW64 /d/dev/go/example
$ ls -l src/sailing
total 2
-rw-r--r-- 1 Randy 197609  95 Apr  8 18:55 power.go
-rw-r--r-- 1 Randy 197609 400 Apr  8 21:53 sail.go
```

Packages

- Packages in Go act like libraries or modules in other languages
- The source code for a package resides in one or more .go files in a directory whose name ends with the import path
  - prometheus/discovery/discovery.go
  - stored in $GOPATH/src/github.com/prometheur/prometheus/discovery
- The GOPATH environment variable is used to specify the Go Workspace
  - The Go workspace is the directory under which projects and packages are located
  - Even though the GOPATH may be a list of directories, it is generally set to a single folder for all Go code on your machine
  - Subdirectories are searched as needed
- Each package serves as a separate namespace
  - To refer to a function from outside its package, qualify the identifier with the package name
  - discovery.ProvidersFromConfig()
- The comment at the top of a package file serves as the package documentation
  - Only one file in each package should have a package doc comment
  - Extensive doc comments are typically placed in a file of their own called doc.go by convention

Package Names

- When a package is imported the package name becomes an accessor for the contents
  - import “test"
- The package name should be:
  - Short
  - Concise
  - Evocative
- By convention packages are given lower case single-word names
  - No need for underscores or mixedCaps
  - Err on the side of brevity, everyone using your package will be typing that name
- Don't worry about collisions
  - The package name is only the default name for imports
  - The package name need not be unique across all source code
  - In case of a collision the importing package can choose a different name to use locally
- Another convention is that the package name is the base name of its source directory
  - The package in src/encoding/base64 is imported as "encoding/base64" but has name “base64”
- Don't use the ‘import .’ Notation
  - This can simplify tests that must run outside the package they are testing, but should otherwise be avoided
- Consider the package name when naming package elements
  - The buffered reader type in the bufio package is called Reader, not BufReader
  - Users see it as bufio.Reader
  - bufio.Reader does not conflict with io.Reader
  - Use the package structure to help you choose good names
  - once.Do; once.Do(setup) reads well and would not be improved by writing once.DoOrWaitUntilDone(setup)
    - Long names don't automatically make things more readable
    - A helpful doc comment can often be more valuable than an extra long name

Package import and initialization

- It is an error to refer to an external element without importing its package
- It is an error to import a package and not refer to it
- The goimports tool will automatically configure an application's import statements
  - Install via package manager or with go get
  - go get golang.org/x/tools/cmd/goimports
- Packages initialization
  - Package variables are initialized in the order declared
    - dependencies are resolved first however
  - Files in a Package are processed in lexical order
- Any file may contain any number of init functions
  - func init() { /* ... */ }
  - init functions can't be called or referenced
  - init functions are automatically executed when the program starts
    - in the order in which they are declared

```
user@ubuntu:~/go/src/lab03$ cat lab03.go
package main

func main() {
    fmt.Println("Hi")
}

user@ubuntu:~/go/src/lab03$ goimports lab03.go > lab03b.go
user@ubuntu:~/go/src/lab03$ cat lab03b.go
package main

import "fmt"

func main() {
        fmt.Println("Hi")
}
```

Scope

- Syntactic block
  - A sequence of statements enclosed in braces
  - Identifiers declared within a syntactic block are not visible outside that block
- Lexical blocks
  - The entire source code
    - the universe block
  - Each package
  - Each file
  - Each for
  - Each if
  - Each switch
    - Also for each case in a switch or select statement
  - Each explicit syntactic block
- A declaration's lexical block determines its scope
  - Builtins live in the Universe Block
  - Declarations outside any function (at package level) can be referred to from any file in the same package
  - Imported packages are declared at the file level, so they can be referred to from the same file
  - Local declarations can be referred to only from within the same block

Inner declarations shadow (hide) outer declarations

example.go

```
1  package main
2
3  import "fmt"
4  import "sailing"
5
6  func main() {
7      fmt.Println(sailing.CalcFT(12.7, 18.9))
8  }
9
```

sail.go

```
1  package sailing
2
3  //CalcM returns the main sail area
4  func CalcM(e float32, p float32) float32 {
5      return e * p / 2
6  }
7
8  //CalcFT returns the fore triangle (jib sail area)
9  func CalcFT(j float32, i float32) float32 {
10     return j * i / x
11 }
12
13 //CalcSailArea returns the total sail area
14 func CalcSailArea(e float32, p float32, j float32, i float32) float32 {
15     return CalcM(e, p) + CalcFT(j, i)
16 }
17
```

power.go

```
1  package sailing
2
3  const x = 2.0
4
5  var y int = 8
6
7  func test() float32 {
8      return x
9  }
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
API server listening at: 127.0.0.1:34218
120.015
```

go get

- The default $GOPATH on *nix is $HOME/go
  - Many store programs there
- The "go get" subcommand allows you to install packages from the internet on your GOPATH
  - $ go get github.com/google/codesearch/index
  - $ go get github.com/petar/GoLLRB/llrb
  - The specified projects are downloaded and installed into $HOME/go
    - src/github.com/google/codesearch/index/
    - src/github.com/petar/GoLLRB/llrb/
    - The compiled packages and their dependencies are placed in pkg/

```
user@ubuntu:~/go/lab01$ ll ../src
total 8
drwxrwxr-x 2 user user 4096 Apr  9 00:23 ./
drwxrwxr-x 4 user user 4096 Apr  9 00:17 ../
user@ubuntu:~/go/lab01$ go get github.com/golang/example/hello
user@ubuntu:~/go/lab01$ ll ../src
total 12
drwxrwxr-x 3 user user 4096 Apr  9 00:23 ./
drwxrwxr-x 6 user user 4096 Apr  9 00:23 ../
drwxrwxr-x 3 user user 4096 Apr  9 00:23 github.com/
user@ubuntu:~/go/lab01$ ll ../src/github.com/
total 12
drwxrwxr-x 3 user user 4096 Apr  9 00:23 ./
drwxrwxr-x 3 user user 4096 Apr  9 00:23 ../
drwxrwxr-x 3 user user 4096 Apr  9 00:23 golang/
user@ubuntu:~/go/lab01$ ll ../src/github.com/golang/
total 12
drwxrwxr-x 3 user user 4096 Apr  9 00:23 ./
drwxrwxr-x 3 user user 4096 Apr  9 00:23 ../
drwxrwxr-x 9 user user 4096 Apr  9 00:23 example/
user@ubuntu:~/go/lab01$ ll ../src/github.com/golang/example/
total 52
drwxrwxr-x  9 user user  4096 Apr  9 00:23 ./
drwxrwxr-x  3 user user  4096 Apr  9 00:23 ../
drwxrwxr-x  3 user user  4096 Apr  9 00:23 appengine-hello/
drwxrwxr-x  8 user user  4096 Apr  9 00:23 .git/
drwxrwxr-x 12 user user  4096 Apr  9 00:23 gotypes/
drwxrwxr-x  2 user user  4096 Apr  9 00:23 hello/
-rw-rw-r--  1 user user 11358 Apr  9 00:23 LICENSE
drwxrwxr-x  2 user user  4096 Apr  9 00:23 outyet/
-rw-rw-r--  1 user user  2634 Apr  9 00:23 README.md
drwxrwxr-x  2 user user  4096 Apr  9 00:23 stringutil/
drwxrwxr-x  2 user user  4096 Apr  9 00:23 template/
user@ubuntu:~/go/lab01$ ll ../src/github.com/golang/example/hello/
total 12
drwxrwxr-x 2 user user 4096 Apr  9 00:23 ./
drwxrwxr-x 9 user user 4096 Apr  9 00:23 ../
-rw-rw-r-- 1 user user  706 Apr  9 00:23 hello.go
user@ubuntu:~/go/lab01$
```

go get tools

- The go get fetching of source code is done by using one of the following tools expected to be found on your system:
  - svn - Subversion, download at: http://subversion.apache.org/packages.html
  - hg - Mercurial, download at https://www.mercurial-scm.org/downloads
  - git - Git, download at http://git-scm.com/downloads
  - bzr - Bazaar, download at http://wiki.bazaar.canonical.com/Download
- For example, git is used for Github, hg is used for Bitbucket, etc.
- For setting proxies for these tools, look here:
  - https://github.com/golang/go/wiki/GoGetProxyConfig

Identifiers

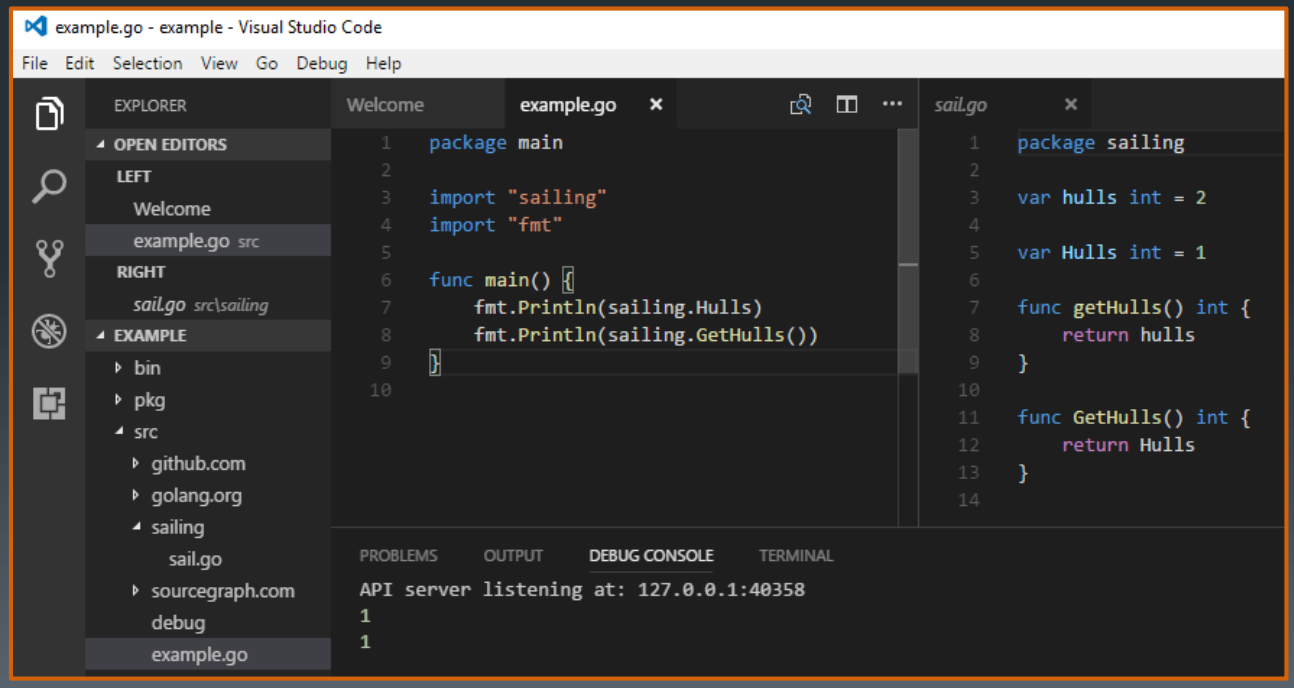
- Go identifiers are used to reference variables, types and other user defined things
- Identifiers are case sensitive
- Identifiers must begin with a Unicode letter or an underbar
  - Unicode divides characters into several major categories: Letter, Mark, Number, Punctuation, Symbol, Separator and Other
- After the first character Names can container any number of letters, underbars and/or digits (there is no limit on name length)
- Idiomatic Go uses camel case
  - Acronyms are always same case (e.g. HTML or html, not Html)
- Keywords cannot be used as identifiers, the 25 go keywords are:

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

- Predeclared names can be used as identifiers in some cases yet should be avoided
  - Constants:
    - true false iota nil
  - Types:
    - int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 uintptr float32 float64 complex128 complex64 bool byte rune string error
  - Functions:
    - make len cap new append copy close delete complex real imag panic recover

Identifier Visibility

- If an entity is:
  - Declared within a function it is local to that function
  - Declared outside of a function it is visible in all files of the package
    - If the name also begins with an upper-case letter it is visible externally
      - Such identifiers are said to be exported
      - e.g. Println in the fmt package
- Package names are always lower case
- Using sub-packages for name-spacing is generally not recommended
- Idiomatic Go uses short identifiers for small scopes longer and more meaningful identifiers for larger scopes



Summary

- Go program structure
- Packages
- go get
- Go identifier rules
- Go variables
- Go variable scope and lifetime

Lab: Program Construction and Syntax

- Create more complex programs with various identifiers

5: User defined types

Objectives

- Understand
  - Structs
  - Self referential structs
  - Recursion
  - Struct embedding
  - Encapsulation

Structs

- A struct is an aggregate data type that groups together zero or more named values of arbitrary types as a single entity
  - Each value is called a field
- Fields are usually written one per line
  - Field-name Type
  - Consecutive fields of the same type may be combined
    - ID, Code int32
  - The name of a struct field is exported if it begins with a capital letter
    - A struct type may contain a mixture of exported and unexported fields
- Struct types usually appear within the declaration of a named type like Moto
- If all the fields of a struct are comparable, the struct itself is comparable
  - using == or !=
- A value of a struct type can be written using a struct literal
  - type Point struct{ X, Y int }
    p1 := Point{1, 2} //Fields init in order
    p2 := Point{Y: 2} //X retains 0 value

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       type Moto struct {
7           ID    int64
8           Make  string
9           Model string
10          Price float32
11      }
12      var x Moto
13      x.ID = 12
14      x.Make = "Honda"
15      x.Model = "VFR750"
16      x.Price = 2310.0
17      fmt.Println(x)
18  }
19
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE
2017/04/09 23:00:50 server.go:73:
2017/04/09 23:00:50 debugger.go:68
API server listening at: 127.0.0.1
2017/04/09 23:00:50 debugger.go:41
{12 Honda VFR750 2310}
```

Self referential structs

- A named struct type S can’t declare a field of type S
  - S may declare a field of the pointer type *S
  - This allows for recursive data structures like linked lists and trees
- Go is a call by value language
  - Pointers must be passed to modify most external objects

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/09 23:27:13 server.go:73: Using API v1
2017/04/09 23:27:13 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/09 23:27:13 debugger.go:414: continuing
{100 0xc04204a440 0xc04204a420}
5
55
100
135
178
```

```go
package main

import "fmt"

type node struct {
    id          int64
    left, right *node
}

func insert(root *node, i int64) *node {
    if nil == root {
        return &node{i, nil, nil}
    }
    if i < root.id {
        root.left = insert(root.left, i)
    } else if i > root.id {
        root.right = insert(root.right, i)
    }
    return root
}

func treePrint(root *node) {
    if nil != root.left {
        treePrint(root.left)
    }
    fmt.Println(root.id)
    if nil != root.right {
        treePrint(root.right)
    }
}

func main() {
    root := node{100, nil, nil}
    insert(&root, 178)
    insert(&root, 5)
    insert(&root, 55)
    insert(&root, 135)
    fmt.Println(root)
    treePrint(&root)
}
```

Struct embedding

- Struct embedding allows one named struct type be used as an anonymous field of another struct type
  - Syntactic shortcut where x.f can stand for x.d.e.f
  - Declare a field with a type but no name
    - Called anonymous fields
      - The fields actually have the name of the type but that name can be omitted
    - The type of the field must be a named type or a pointer to a named type
      - type Point struct {X, Y int }
type Circle struct { Point; Radius int }
type Wheel struct { Circle; Spokes int }
      - var w Wheel w.X = 8 //equivalent to w.Circle.Point.X = 8
- Literals follow the shape of the type:
  - w = Wheel{ Circle{ Point{ 8, 8}, 5}, 20}
- Anonymous fields have implicit names so you can't have two anonymous fields of the same type
- Because the name of the field is implicitly determined by its type, so too is the visibility of the field

JSON

- Go supports several encoding libraries
  - encoding/json
  - encoding/xml
- JavaScript Object Notation (JSON) is a standard notation for sending and receiving structured information
- The basic JSON types are
  - numbers (in decimal or scientific notation)
  - booleans (true or false)
  - strings
- These basic types may be combined recursively using JSON arrays and objects
  - JSON array is an ordered sequence of values, written as a comma-separated list enclosed in square brackets
    - JSON arrays are used to encode Go arrays and slices
  - JSON object is a mapping from strings to values, written as a sequence of name:value pairs separated by commas and surrounded by braces
    - JSON objects are used to encode Go maps (with string keys) and structs
- The json.Marshall() method accepts a Go object and returns its JSON representation
  - MarshalIndent() provides human friendly formatting
  - Only exported fields are marshaled

```go
package main

import "fmt"
import "encoding/json"

type Moto struct {
    ID          int64
    Make, Model string
}

func main() {
    b1 := Moto{34, "Honda", "VFR750"}
    data, _ := json.Marshal(b1)
    fmt.Printf("%s\n", data)
    data, _ = json.MarshalIndent(b1, "", "    ")
    fmt.Printf("%s\n", data)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 00:18:40 server.go:73: Using API v1
2017/04/10 00:18:40 debugger.go:68: launching process w
API server listening at: 127.0.0.1:2345
2017/04/10 00:18:40 debugger.go:414: continuing
{"ID":34,"Make":"Honda","Model":"VFR750"}
{
    "ID": 34,
    "Make": "Honda",
    "Model": "VFR750"
}
```

JSON Field Tags

- A field tag is a string of metadata associated with the field of a struct
  - Year int `json:"released"`
- A field tag is conventionally interpreted as a space-separated list of key:"value" pairs
  - since they contain double quotes they are usually written with raw string literals.
- The first part of the JSON field tag specifies an alternative JSON name for the Go field
  - Like total_count for a Go field named TotalCount
- Additional option, omitempty, which indicates that no JSON output should be produced if the field has the zero value
  - Comma separated from the alt field name but within the double quotes

```go
package main

import "fmt"
import "encoding/json"

type Moto struct {
    ID          int64 `json:"model_no"`
    Make, Model string
}

func main() {
    b1 := Moto{34, "Honda", "VFR750"}
    data, _ := json.Marshal(b1)
    fmt.Printf("%s\n", data)
    data, _ = json.MarshalIndent(b1, "", "    ")
    fmt.Printf("%s\n", data)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 00:30:50 server.go:73: Using API v1
2017/04/10 00:30:50 debugger.go:68: launching process
API server listening at: 127.0.0.1:2345
2017/04/10 00:30:50 debugger.go:414: continuing
{"model_no":34,"Make":"Honda","Model":"VFR750"}
{
    "model_no": 34,
    "Make": "Honda",
    "Model": "VFR750"
}
```

Unmarshalling

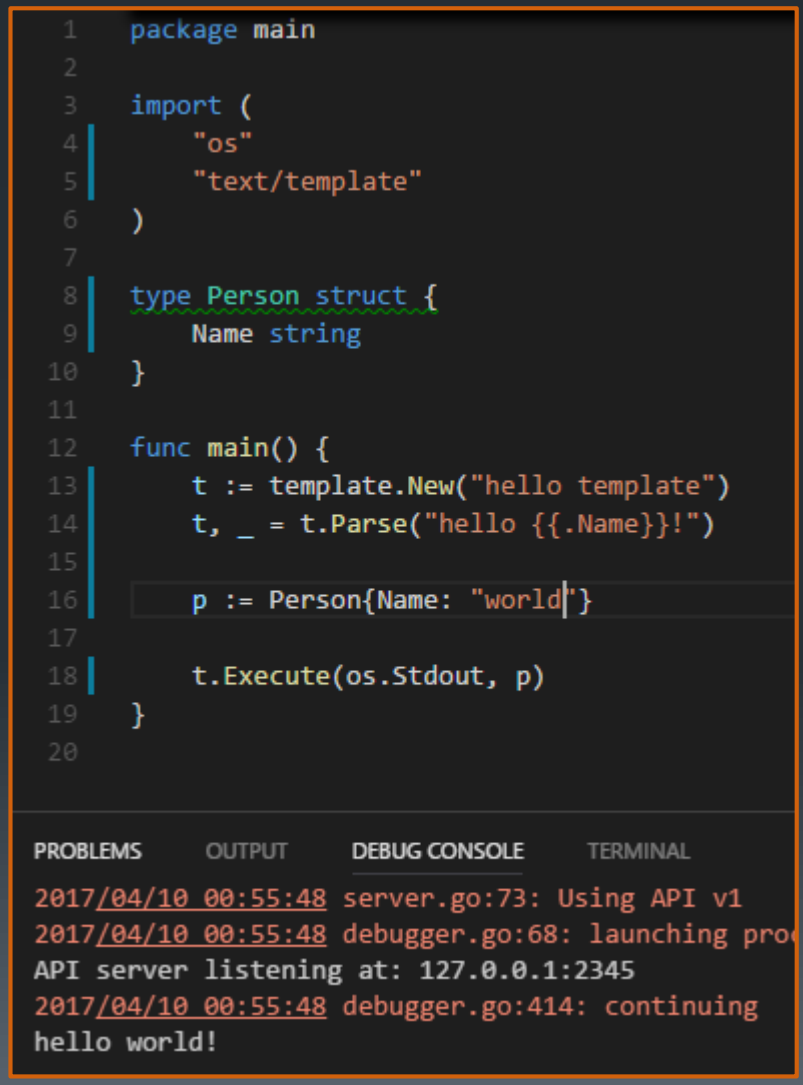
- JSON text can be unmarshalled back into Go structures
  - json.Unmarshal()
  - Unmarshal takes the text to unmarshal and a point to the struct to unmarshal into
  - If the operation fails an error is returned
- The unmarshal operation will populate the elements of the Go struct that match JSON fields
  - All fields need not be present
  - Unmatched fields retain their zero value
  - Fields to unmarshal to must have initial caps
  - JSON field matching is case insensitive

```go
package main

import "fmt"
import "encoding/json"

type Moto struct {
    ID          int64
    Make, Model string
}

func main() {
    b1 := Moto{34, "Honda", "VFR750"}
    data, _ := json.Marshal(b1)
    fmt.Printf("%s\n", data)
    var modelInfo struct {
        ID    int64
        Model string
    }
    json.Unmarshal(data, &modelInfo)
    fmt.Println(modelInfo)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 00:37:56 server.go:73: Using API v1
2017/04/10 00:37:56 debugger.go:68: launching p
API server listening at: 127.0.0.1:2345
2017/04/10 00:37:57 debugger.go:414: continuing
{"ID":34,"Make":"Honda","Model":"VFR750"}
{34 VFR750}
```

Templates

- Go offers several packages which provide a mechanism for substituting the values of variables into a template
  - text/ template
  - html/ template
- A template is a string or file containing one or more portions enclosed in double braces: {{...}}
  - Called actions
  - Each action contains an expression in the template language
    - printing values
    - selecting struct fields
    - calling functions
    - expressing control flow (if-else/range loops)
    - Even instantiating other templates



Summary

- Go includes a range of features for creating UDTs
  - Structs
  - Self referential structs
  - Recursion
  - Struct embedding
  - Encapsulation

Lab: User Defined Types

- Working with structs

6: First Class Functions

Objectives

- Gain an deeper understanding of Go functions
- Understand:
  - Anonymous functions
  - Variadic functions
  - Closures

Go Functions

- A function declaration has
  - The keyword ‘func’
  - A name
  - A list of parameters
  - An optional list of results
  - A body

- A Function’s signature is its type, the list of parameter and return types it takes/produces (ignoring names)

```
func functionName(paramName paramType) returnType {
        //body
}
```

Function Parameters

- The parameter list specifies the names and types of the function's parameters
- These are the local variables whose values or arguments are supplied by the called
- The parameter name is followed by the type
- If two or more consecutive named functions parameters share the same type, the type only needs to follow the last parameter
- The blank identifier (underbar) can be used to accept parameters that will be discarded

```
func add(x int, y int, z int) {
    return x + y + z
}

func add(x, y, z int) {
    return x + y + z
}
```

Function Results

- The result list specifies the types of the values that the function returns
- There can be multiple return values
- If the function returns one unnamed result or no results at all, parentheses are optional and usually omitted
- Leaving off the result list entirely declares a function that does not return any value and is called only for its effects
- Results can be named
  - Each name declares a local variable initialized to the zero value for its type
  - Very useful when returning multiple values of the same type

```
1   package main
2
3   import "fmt"
4
5   func sub(x, y int) (z int) {
6       z = x - y
7       return
8   }
9
10  func main() {
11      fmt.Println(sub(8, 4))
12  }
13
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
2017/04/10 14:19:37 server.go:73: Using API v1
2017/04/10 14:19:37 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/10 14:19:37 debugger.go:414: continuing
4
```

Go Functions

- Callers must provide an argument for each parameter
  - Arguments must be in the order the parameters are declared
  - Go has no concept of default parameter values
  - Go offers no way to specify arguments by name
- Parameters are local variables within the body of the function
  - Their initial values are set to the arguments supplied by the caller
  - Function parameters and named results are variables in the function's outermost lexical block
  - Arguments are passed by value
    - Functions receive a copy of each argument
    - Modifications to the copy do not affect the caller
    - If the argument contains a reference (pointer, slice, map, function, or channel) the caller may be affected by modifications the function makes to external objects indirectly
- Functions without a body indicate the function is implemented in a language other than Go
- Functions may be recursive
  - They may call themselves directly or indirectly
  - This can give rise to stack overflows in some languages, for example, when traversing deep data structures
  - Go implementations use variable-size stacks that start small and grow as needed
    - Up to a limit on the order of a gigabyte
    - This makes recursion fairly safe and causes stack overflow event to fall into the clearly erroneous category in most cases
- Named functions can be declared only at the package level
  - Function literals (lamdas) can denote a function value within any expression

First Class Functions

- Functions are first-class values in Go
  - function values have types
  - may be assigned to variables
  - may be passed or returned from functions
  - may be called like any other function
  - are NOT comparable
  - can NOT be used as map keys
- Cause a panic when unset and invoked

```go
package main

import "fmt"

func afun() {
    fmt.Println("A")
}

func bfun() {
    fmt.Println("B")
}

func main() {
    f := afun
    f()
    f = bfun
    f()
    var g func(int32) float64
    g(3)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 17:28:58 server.go:73: Using API v1
2017/04/10 17:28:58 debugger.go:68: launching process with args: [d:\dev
API server listening at: 127.0.0.1:2345
2017/04/10 17:28:58 debugger.go:414: continuing
A
B
2017/04/10 17:29:05 debugger.go:414: continuing
panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xc0000005 code=0x0 addr=0x0 pc=0x488bb0]

goroutine 1 [running]:
main.main()
        d:/dev/go/example/src/example.go:19 +0x60
```

Variadic Functions

- Variadic functions can be called with varying numbers of arguments
  - fmt.Printf requires one fixed argument then accepts any number of subsequent arguments
- To declare a variadic function the type of the final parameter is preceded by an ellipsis: "...int"
  - This implicitly packs the parameters into a slice of that type
    - The caller allocates an array, copies the arguments into it, and passes a slice of the entire array to the function
  - This can also be done explicitly
    - Pass a slice with a trailing ellipsis
    - To unpack a sliced array: a[:]…
- The type of a variadic function is different from the identical function that accepts a slice explicitly
- To create a variadic function that can accept any type in any position make the last argument an interface type
  - More on this in the next module

```go
package main

import "fmt"

func sum(v ...int) int {
    total := 0
    for _, x := range v {
        total += x
    }
    return total
}

func main() {
    fmt.Println(sum(0))
    fmt.Println(sum(1))
    fmt.Println(sum(1, 2))
    fmt.Println(sum(1, 2, 3))
    s := []int{4, 5, 6}
    fmt.Println(sum(s[:]...))
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 20:07:14 server.go:73: Using API v1
2017/04/10 20:07:14 debugger.go:68: launching pr
API server listening at: 127.0.0.1:2345
2017/04/10 20:07:14 debugger.go:414: continuing
0
1
3
6
15
```

Library Functions values

- The Go standard library makes extensive use of function values
  - strings.Map()
  - bytes.Map()
- Function literals can be convenient in contexts where a simple function must be supplied as a parameter (line 14)

```
1   package main
2
3   import (
4       "fmt"
5       "strings"
6   )
7
8   func inc(r rune) rune {
9       return r + 1
10  }
11
12  func main() {
13      fmt.Println(strings.Map(inc, "0123456"))
14      fmt.Println(strings.Map(func(r rune) rune { return r + 2 }, "0123456"))
15  }
16
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 17:40:58 server.go:73: Using API v1
2017/04/10 17:40:58 debugger.go:68: launching process with args: [d:\dev\go\example
API server listening at: 127.0.0.1:2345
2017/04/10 17:40:58 debugger.go:414: continuing
1234567
2345678
```

  - A function literal is written like a function declaration, but without a name following the func keyword
  - It is an expression whose value is called an anonymous function
  - Function literals let us define a function at its point of use

Summary

- Go provides
  - Many common function features
    - Parameters, return values, recursion, …
  - Several not so common function features implemented in a Go specific way
    - Anonymous functions/lamdas, Closures, …

Lab: Functions

- Go specific function features

7: Methods and Interfaces

Objectives

- Explore:
  - Methods
  - Receivers
  - Method binding
  - Encapsulation
  - Interfaces
  - Type Assertions

Methods

- There is no universal definition of object-oriented programming
- In Go, an object is a variable that has methods
- A method is a function associated with a particular type
  - An object-oriented program is one that uses methods to express the properties and operations of each data structure so that clients need not access the object's representation directly
  - This is one of the key features of object orientation: Encapsulation
- A method is declared much like an ordinary function but with an extra parameter before the function name
  - This parameter attaches the function to the type
  - The type parameter name is called the receiver and is like the "this" or "self" pointer in some other OO languages
    - receiver names are commonly the first letter of the type name
- Methods are invoked using . notation on the desired object: bike.mm()
  - Such and expression is called a selector in Go
- Each type has its own namespace allowing multiple types to have the same method names
  - Methods can also use names already taken at the package level
  - This allows method names to be compact
- Methods may be declared on any named type defined in the same package, so long as its underlying type is neither a pointer nor an interface

```go
package main

import "fmt"

type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

type digit int

func (i digit) prt() string {
    return fmt.Sprintf("int: %d", i)
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    fmt.Println(bike.mm())

    var x digit = 7
    fmt.Println(x.prt())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:05:30 server.go:73: Using API v1
2017/04/10 22:05:30 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/10 22:05:31 debugger.go:414: continuing
Honda VFR750
int: 7
```

Pointer Receivers

- Calling a function makes a copy of each argument value
- If a function needs to update a variable or if an argument is large, passing the address of the variable using a pointer is desirable
- Methods that need to update the receiver variable need a pointer receiver
  - Even when pointer receivers are defined methods can be called with the name directly
    - bike.ModelSuffix()
      - As opposed to:
    - (&bike).ModelSuffix()
- Nil can be a legal receiver value for types with a nil zero value

```go
package main

import "fmt"

// Moto is for motorcycle
type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

func (m *Moto) ModelSuffix(suf string) {
    m.Model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    fmt.Println(bike.mm())
    bike.ModelSuffix("F")
    fmt.Println(bike.mm())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:16:56 server.go:73: Using API v1
2017/04/10 22:16:56 debugger.go:68: launching proc
API server listening at: 127.0.0.1:2345
2017/04/10 22:16:56 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
```

Binding Methods

- It's possible to separate method/receiver binding from invocation
- The selector p.mm yields a function with the method mm bound to variable p
  - This function can then be invoked without a receiver value
    - x := p.mm
    - x()
- The selector Moto.mm yields a function invoking the method mm with an additional parameter prepended to accept the receiver
  - x := Moto.mm
  - x(bike)
- Binding methods and objects is shockingly simple compared to the process in other languages (C++98 anyone?)

```go
package main

import "fmt"

// Moto is for motorcycle
type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

func (m *Moto) ModelSuffix(suf string) {
    m.Model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}

    bmm := Moto.mm
    fmt.Println(bmm(bike))

    bms := bike.ModelSuffix
    bms("F")
    fmt.Println(bike.mm())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:37:49 server.go:73: Using API v1
2017/04/10 22:37:49 debugger.go:68: launching proce
API server listening at: 127.0.0.1:2345
2017/04/10 22:37:50 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
```

Encapsulation in Go

- In Go the unit of encapsulation is the package
  - NOT the type
  - The fields of a struct type are visible to all code within the same package
- Go has only one mechanism to control the visibility of names:
  - Identifier Capitalization
    - Capitalized names are exported from the package
    - Uncapitalized names are not exported from the package
  - This rule also applies to the fields of a struct or the methods of a type
- To encapsulate an object we must make it a struct and give the attribute(s) lowercase names
  - It is common to have structs with a single field for this reason
- Encapsulation provides three benefits
  - one need inspect fewer statements to understand the possible values of variables
  - hiding implementation details prevents clients from depending on things that might change
  - clients are prevented from setting an object's variables arbitrarily

```go
package main

import "fmt"

type Moto struct {
    make  string
    model string
}

func (m Moto) mm() string {
    return m.make + " " + m.model
}

func (m *Moto) ModelSuffix(suf string) {
    m.model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}

    bmm := Moto.mm
    fmt.Println(bmm(bike))

    bms := bike.ModelSuffix
    bms("F")
    fmt.Println(bike.mm())
    fmt.Println(bike.make) //we in the same package so this works!
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:50:37 server.go:73: Using API v1
2017/04/10 22:50:37 debugger.go:68: launching process with args: [d:\dev\g
API server listening at: 127.0.0.1:2345
2017/04/10 22:50:37 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
Honda
```

Interfaces

- Interface types express generalizations or abstractions about the behaviors of other types
  - By generalizing, interfaces let us write functions that are more flexible and adaptable because they are not tied to the details of one particular implementation
- Go's interfaces are satisfied implicitly
  - There's no need to declare interfaces that a given type satisfies
  - Simply possessing the necessary methods is enough
  - This is a fairly unique approach to interface/type association
- This design lets you create new interfaces that are satisfied by old types without changing the types
  - particularly useful for types defined in packages that you don't control
- A concrete type specifies the exact representation of its values and exposes the intrinsic operations of that representation
- An Interface type is an abstract type and doesn't expose the representation or internal structure of its values, or the set of basic operations supported
  - It reveals only some of their methods
  - When you have a value of an interface type, you know nothing about what it is; you know only what behaviors are provided by its methods
- The io package offers several key interfaces

type **ByteReader**

ByteReader is the interface that wraps the ReadByte method.

ReadByte reads and returns the next byte from the input.

```
type ByteReader interface {
        ReadByte() (byte, error)
}
```

type **ByteScanner**

ByteScanner is the interface that adds the UnreadByte method to the basic ReadByte method.

UnreadByte causes the next call to ReadByte to return the same byte as the previous call to ReadByte. It may be an error to call UnreadByte twice without an intervening call to ReadByte.

```
type ByteScanner interface {
        ByteReader
        UnreadByte() error
}
```

type **ByteWriter**

ByteWriter is the interface that wraps the WriteByte method.

```
type ByteWriter interface {
        WriteByte(c byte) error
}
```

type **Closer**

Closer is the interface that wraps the basic Close method.

The behavior of Close after the first call is undefined. Specific implementations may document their own behavior.

```
type Closer interface {
        Close() error
}
```

Interface Combinations

- An interface type specifies a set of methods that a concrete type must possess to be considered an instance of that interface
  - A type satisfies an interface if it possesses all the methods the interface requires.
- The io.Writer type is one of the most widely used interfaces because it provides an abstraction of all the types to which bytes can be written
  - Files
  - Memory buffers
  - Network conns
  - HTTP clients
  - Archivers
  - Hashers
- The io package defines interface types in terms of other interfaces as well

type **WriteCloser**

WriteCloser is the interface that groups the basic Write and Close methods.

```
type WriteCloser interface {
        Writer
        Closer
}
```

type **WriteSeeker**

WriteSeeker is the interface that groups the basic Write and Seek methods.

```
type WriteSeeker interface {
        Writer
        Seeker
}
```

type **Writer**

Writer is the interface that wraps the basic Write method.

Write writes len(p) bytes from p to the underlying data stream. It returns the number of bytes written from p (0 <= n <= len(p)) and any error encountered that caused the write to stop early. Write must return a non-nil error if it returns n < len(p). Write must not modify the slice data, even temporarily.

Implementations must not retain p.

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

Implementing an interface

- Imagine you would like to use the fmt.Fprintf method to write formatted text to a buffering type you have
  - The help for Fprintf says it wants an io.Writer which only has one method
  - Implementing that method will satisfy the interface and make it possible for you to use that type with many library functions
- Note that our receiver is a pointer
  - We need to modify the object
- Because of this we must pass the pointer to the object (&b) to functions desiring a Writer interface

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

```
1  package main
2
3  import "fmt"
4
5  type StrBuf struct {
6      buf string
7  }
8
9  func (sb *StrBuf) Write(p []byte) (n int, err error) {
10     sb.buf = string(p)
11     return len(sb.buf), nil
12 }
13
14 func main() {
15     var b StrBuf
16     fmt.Fprintf(&b, "Hi")
17     fmt.Println(b.buf)
18 }
19
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 23:50:05 server.go:73: Using API v1
2017/04/10 23:50:05 debugger.go:68: launching process with args:
API server listening at: 127.0.0.1:2345
2017/04/10 23:50:05 debugger.go:414: continuing
Hi
```

func **Fprintf**

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

Fprintf formats according to a format specifier and writes to w. It returns the number of bytes written and any write error encountered.

Custom Interfaces

- Interface types are defined with the interface keyword
- Types implement interfaces by offering methods with identical signatures as those in the interface
- Interface variables can point to any object implementing the interface
  - This sets the type and value of the interface variable
    - Known as the dynamic type and value
      - Together these are referred to as the type descriptors
    - The static type of an interface variable is that of the interface
- nil interface calls cause a panic
  - if i == nil //test for unassigned interface variable
- Interface values may be compared using == and !=
  - Two interface values are equal if both are nil, or if their dynamic types are identical and their dynamic values are equal
  - Being comparable, interfaces can be used as map keys

```go
package main

import "fmt"

type Moto struct {
    make, model string
}

func (m Moto) Summary() string {
    return m.make + " " + m.model
}

type Veh interface {
    Summary() string
}

func main() {
    bike1 := Moto{"Honda", "VFR750"}
    bike2 := Moto{"Ducati", "Paso750"}
    var i Veh
    i = bike1
    fmt.Println(i.Summary())
    i = bike2
    fmt.Println(i.Summary())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/11 00:14:54 server.go:73: Using API v1
2017/04/11 00:14:54 debugger.go:68: launching pr
API server listening at: 127.0.0.1:2345
2017/04/11 00:14:54 debugger.go:414: continuing
Honda VFR750
Ducati Paso750
```

Type Assertion

- Type assertions in Go allow you to test the types supported by the dynamic type associated with an interface
  - Form: i.(T)
    - Where i is an interface and T is the desired type
  - Type assertions return a tuple with the desired interface type and an ok status
    - i2, ok := i.(T)
    - The type tested need NOT be the current dynamic type of i or a component thereof
- Much like dynamic casting in other languages
  - A key difference is that the interface supplied and the interface derived need not be related in any way

```go
package main

import "fmt"

type Moto struct {
    make, model string
}

func (m Moto) Summary() string {
    return m.make + " " + m.model
}

type Veh interface {
    Summary() string
}

type Car interface {
    Drive() string
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    var i Veh
    i = bike
    if v, ok := i.(Veh); ok {
        fmt.Println(v.Summary())
    }
    if c, ok := i.(Car); ok {
        fmt.Println(c.Drive())
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/11 00:32:57 server.go:73: Using API v1
2017/04/11 00:32:57 debugger.go:68: launching proc
API server listening at: 127.0.0.1:2345
2017/04/11 00:32:57 debugger.go:414: continuing
Honda VFR750
```

Go Tips

- When designing a new package:
  - Use Interfaces when two or more concrete types must be dealt with in a uniform way
  - Avoid single implementation interfaces
    - These are unnecessary abstractions and have a run-time cost
    - The exception is a single type interface that cannot live in the same package as the type because of dependencies (an interface can decouple the two packages)
  - Keep interfaces minimal, crisp and generic
    - Because interfaces are used when implemented by multiple types they tend to have fewer, simpler methods (often just one)
    - Small interfaces are easier to satisfy when new types come along
    - A good rule of thumb for interface design is ask only for what you need
  - Control method visibility outside a package through exporting

Source: Donovan/Kernighan

Summary

- Go support several key OO features:
  - Methods
  - Receivers
  - Method binding
  - Encapsulation
  - Interfaces
  - Type Assertions

Lab: Methods and Interfaces

- Working with methods and interfaces

8: Error handling

Objectives

- Explore Go error management
- Understand:
  - Errors
  - panic()
  - recover()
  - defer

Defer

- defer statement
  - An ordinary function call prefixed by the defer keyword
- Deferred function and argument expressions are evaluated when the statement is executed
- The actual call is deferred until the function completes
  - By return statement, falling off the end, exiting or panicking
- Any number of calls may be deferred
  - Deferred calls are executed in reverse order in which they were deferred
- A defer statement is often used with paired operations
  - e.g. open/close, connect/disconnect, lock/unlock
  - Performs like a finally block in some languages, ensuring that resources are released in all cases
  - The right place for a defer statement is immediately after the resource has been successfully acquired
- Deferred functions run after return statements have updated the function's result variables
  - An anonymous function can access its enclosing function's variables, including named results, even change them … (!)
- The example program defers the HTTP response body close operation
  - Also demonstrates use of nested anonymous functions for internal recursion
  - This function uses a closure to call itself (required since the function is anonymous)
  - The code uses the golang net/html extension package (not part of the std lib) to parse the HTML document

```go
package main

import (
    "fmt"
    "net/http"
    "strings"

    "golang.org/x/net/html"
)

func title(url string) (*html.Node, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return nil, fmt.Errorf("% s has type %s, not text/html", url, ct)
    }
    doc, err := html.Parse(resp.Body)
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    var f func(n *html.Node) *html.Node
    f = func(n *html.Node) *html.Node {
        if n.Type == html.ElementNode && n.Data == "title" {
            return n.FirstChild
        }
        for c := n.FirstChild; c != nil; c = c.NextSibling {
            p := f(c)
            if p != nil {
                return p
            }
        }
        return nil
    }
    return f(doc), nil
}

func main() {
    el, _ := title("http://rx-m.com")
    fmt.Println(el.Data)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 20:51:43 server.go:73: Using API v1
2017/04/10 20:51:43 debugger.go:68: launching process with args: [d:\dev\go\exampl
API server listening at: 127.0.0.1:2345
2017/04/10 20:51:43 debugger.go:414: continuing
RX-M - DevOps & Cloud Native Consulting and Training - RX-M
```

Errors

- Some functions always succeed (200)
- Some functions fail if preconditions are not met (400)
  - Errors are a result of bad calls
- Some functions given proper preconditions fail due to factors beyond the functions control (500)
  - Any function that does I/O
- Errors are an important part of a package's API
  - Go considers errors one of a set of expected behaviors
  - If a function can fail it is expected to return an error result
    - Conventionally the last thing in the return list
  - If the failure has only one possible cause the result is (by convention) a Boolean called "ok"

```go
package main

import "fmt"

func div(x, y int) (res int, ok bool) {
    if y == 0 {
        return 0, false
    }
    return x / y, true
}

func main() {
    ans, ok := div(8, 4)
    if ok {
        fmt.Println("First try: ", ans)
    }
    ans, ok = div(8, 0)
    if ok {
        fmt.Println("Second try: ", ans)
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 15:11:57 server.go:73: Using API v1
2017/04/10 15:11:57 debugger.go:68: launching pr
API server listening at: 127.0.0.1:2345
2017/04/10 15:11:57 debugger.go:414: continuing
First try:  2
```

error

- Non trivial (bool) errors are indicated by returning an “error” instance as a final return value
- error is a built-in interface type
  - An error may be:
    - nil - implies success
    - non-nil - implies failure
      - Such an error has a message string obtained by calling its Error method
      - To print the message: fmt.Println(err) or fmt.Printf("%v", err) of fmt.Println(err.Error())
      - Usually when a function returns a non-nil error its other results are undefined
        - Some rare functions return partial results in error cases
- Go does not offer a traditional exception handling system
  - Go supports a type of exception for reporting truly unexpected errors that indicate a bug (not to be used for reporting routine errors)
  - The Go philosophy is that exceptions entangle the description of an error with the control flow required to handle it
    - This leads to routine errors being reported to the end user in a stack trace, full of information about the structure of the program but lacking intelligible context about what went wrong
  - Go programs use ordinary control-flow mechanisms (e.g. if and return) to respond to errors
    - This style undeniably demands that more attention be paid to error-handling logic
    - precisely the point

```go
package main

import (
    "errors"
    "fmt"
)

func posdiv(x, y int) (res int, err error) {
    if y < 0 {
        return 0, errors.New("attempted posdiv division with a negative number")
    }
    if y == 0 {
        return 0, errors.New("attempted division by 0")
    }
    return x / y, nil
}

func main() {
    ans, err := posdiv(8, -4)
    if err != nil {
        fmt.Println("Error: ", err.Error())
    } else {
        fmt.Println("Result: ", ans)
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 15:57:03 server.go:73: Using API v1
2017/04/10 15:57:03 debugger.go:68: launching process with args: [d:\dev\go\example\src\
API server listening at: 127.0.0.1:2345
2017/04/10 15:57:03 debugger.go:414: continuing
Error:  attempted posdiv division with a negative number
```

Propagating errors

- Errors in low level routines must often be propagated by mid level routines that have no stake in the error
  - return err
- A call which produces an error such that the callee does not have enough context to produce a meaningful explanation may require the caller to augment it to keep the error meaningful
  - Go allows new errors to be fabricated from lower level error messages
    - Fmt.Errorf("%v", err)
    - errors.Wrap(err, "read failed")
  - Error messages are frequently chained in this way
- Therefore error messages should:
  - Avoid newlines
  - Not be capitalized
  - Provide a meaningful description of the problem
  - Supply sufficient/relevant detail
  - Be consistent
    - errors returned by the same function/package should be of similar form
    - e.g. The os package guarantees that every error returned by a file operation describes the nature of the failure (permission denied) and the name of the file
- Resulting errors may be long, but they will be self-contained and easily parsed by tools like grep

```go
package main

import "errors"
import "fmt"

func posdiv(x, y int) (res int, err error) {
    if y < 0 {
        return 0, errors.New("attempted posdiv division with a negative number")
    }
    if y == 0 {
        return 0, errors.New("attempted division by 0")
    }
    return x / y, nil
}

func multidiv(x, y1, y2 int) (res int, err error) {
    ans1, e1 := posdiv(x, y1)
    ans2, e2 := posdiv(x, y2)
    if e1 != nil || e2 != nil {
        return 0, fmt.Errorf("multidiv component failure, y1: %v, y2: %v", e1, e2)
    }
    return ans1 + ans2, nil
}

func main() {
    ans, err := multidiv(8, 2, -4)
    if err != nil {
        fmt.Println("Error: ", err.Error())
    } else {
        fmt.Println("Result: ", ans)
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 16:09:56 server.go:73: Using API v1
2017/04/10 16:09:56 debugger.go:68: launching process with args: [d:\dev\go\example\src\debug]
API server listening at: 127.0.0.1:2345
2017/04/10 16:09:56 debugger.go:414: continuing
Error:  multidiv component failure, y1: <nil>, y2: attempted posdiv division with a negative number
```

Retry Errors

- Transient/unpredictable problems may require code to retry the failed operation
  - Perhaps with a delay
  - Perhaps with a limit on the number of attempts or time spent
  - Transient errors recovered from should not go unrecorded
- The Go log package
  - Implements simple logging
  - Defines a Logger type with methods for formatting output
  - Has a predefined 'standard' Logger accessible through helper functions Print[f|ln], Fatal[f|ln], and Panic[f|ln]
    - The Fatal functions call os.Exit(1) after writing the log message
    - The Panic functions call panic after writing the log message
  - The standard logger writes to standard error and prints the date and time of each logged message
  - Every log message is output on a separate line
    - if the message being printed does not end in a newline, the logger will add one
- Stderr can be printed to directly with the fmt package
  - fmt.Fprintf(os.Stderr, “big problem!”)

```
1  package main
2
3  import "log"
4
5  func main() {
6      log.Printf("connection failed retrying")
7  }
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/10 16:31:22 server.go:73: Using API v1
2017/04/10 16:31:22 debugger.go:68: launching process
API server listening at: 127.0.0.1:2345
2017/04/10 16:31:23 debugger.go:414: continuing
2017/04/10 16:31:23 connection failed retrying
```

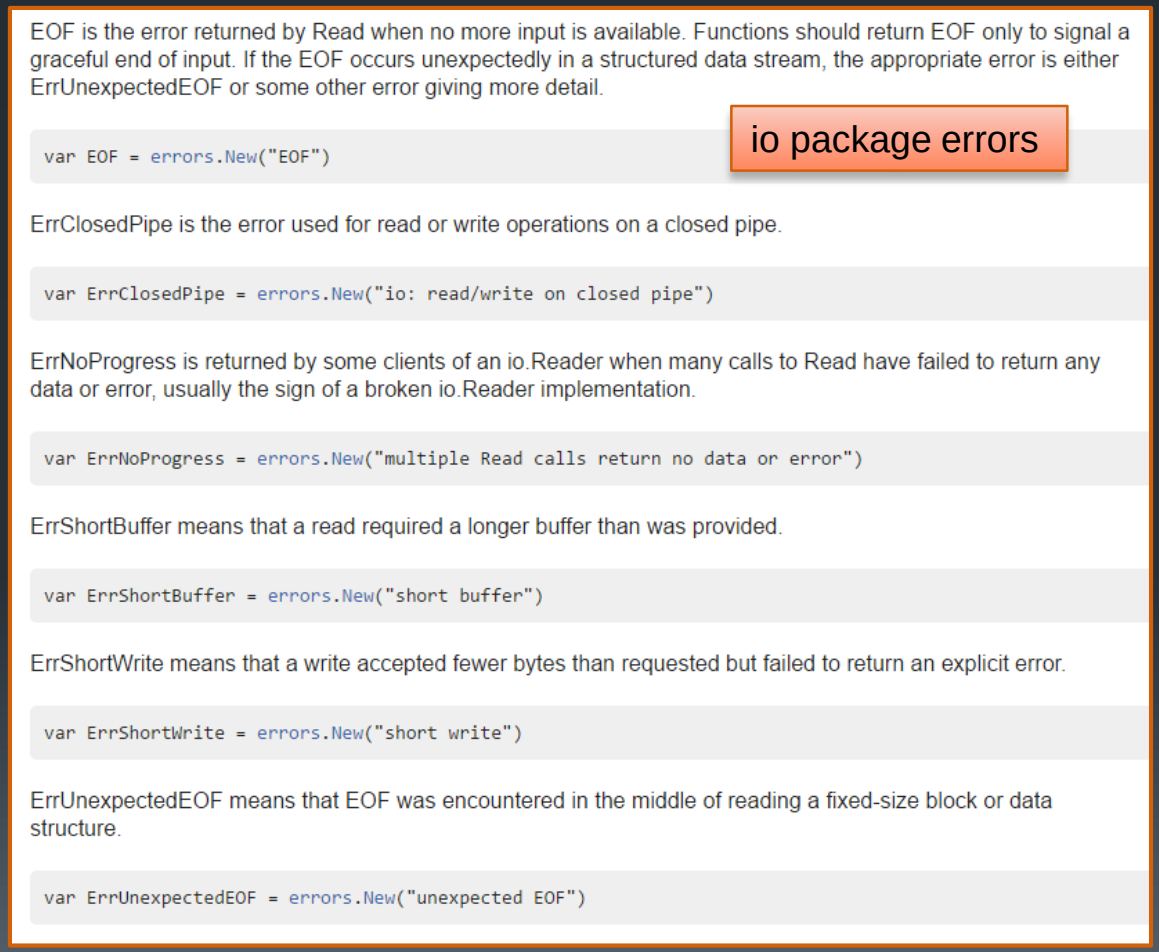
Stopping Execution

- If the error is unrecoverable it may be best to terminate execution
  - The os package provides an func Exit(code int) function to terminate execution
    - The code passed to Exit is passed to the system as the process exit code (0 typically indicates success, all other values indicate failure)
  - This decision should be reserved for the most abstract parts of the program (e.g. main())
- Go error handling style
  - Invoke risky function
    - Checking for error
    - Deal with Failure
      - If failure causes the function to return the success logic follows at the outer level
    - Deal with Success
  - Functions include initial checks to reject errors (DbC*) followed by the substance of the function minimally indented

** In Design by Contract style*

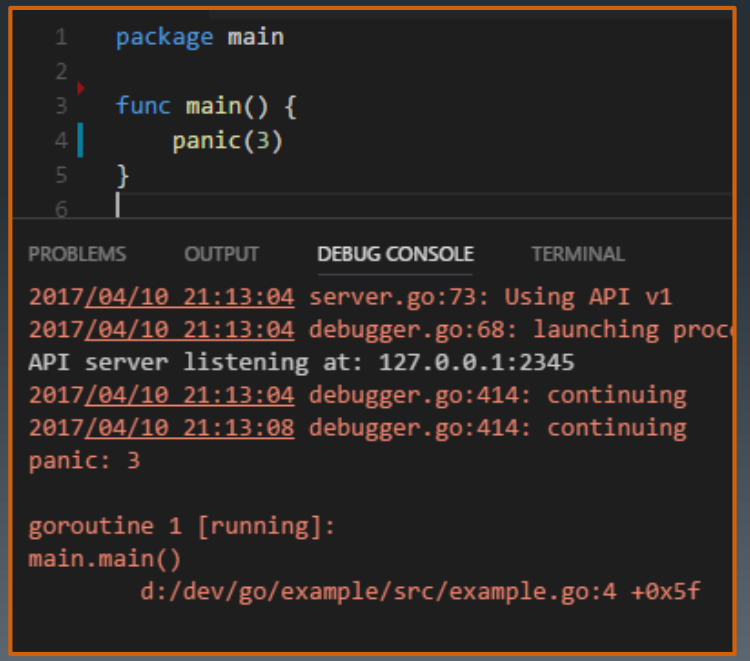
Distinguished Errors

- Some packages define specific errors
  - Referred to as distinguished errors
  - io.EOF is a good example
    - In an end-of-file condition there is no information to report
    - This error has a fixed message: "EOF"
  - if err == io.EOF
    - A legal expression because the io.EOF variable is always returned when an EOF event occurs



Panics

- In Go execution errors such as attempting to index an array out of bounds trigger a run-time panic
  - Can also be triggered by the built-in function panic(string)
- Panics are designed to fail fast, stopping all operations quickly
- A common use of panic is to abort when a function returns an error value that we don't know how to (or want to) handle
  - Last resort, idiomatic go code returns meaningful errors
- During a typical panic, normal execution stops, all deferred function calls are executed, and the program crashes with a log message
  - This log message includes the panic value, which is usually an error message of some sort a stack trace showing the stack of function calls that were active at the time of the panic
  - This log message often has enough information to diagnose the root cause of the problem without running the program again, so it should always be included in a bug report about a panicking program



Recover

- Giving up is usually the right response to a panic
- In some cases it may be possible to recover
  - Or clean up before quitting (e.g. a web server could close connections before crashing)
- If the built-in recover() function is called within a deferred function that is panicking the current state of panic ends and recover() returns the panic code
  - The function that was panicking does not continue where it left off but returns normally
- If recover is called at any other time, it has no effect and returns nil
- The example defers an anonymous function (self invoking due to the trailing parenthesis)
  - The panic code was 3 so the Println() displays this as the result of recover()
  - The program exits normally

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       defer func() {
7           fmt.Println(recover())
8       }()
9       panic(3)
10  }
11
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
2017/04/10 21:22:38 server.go:73: Using API v1
2017/04/10 21:22:38 debugger.go:68: launching proce
API server listening at: 127.0.0.1:2345
2017/04/10 21:22:38 debugger.go:414: continuing
3
```

Summary

- Go provides unique Go specific function features
  - Deferred functions, no exceptions, …
- Go error management is explicit using tools such as:
  - Errors
  - panic()
  - recover()
  - defer

Lab: Error handling

- Error handling

9: Communicating Sequential Processes

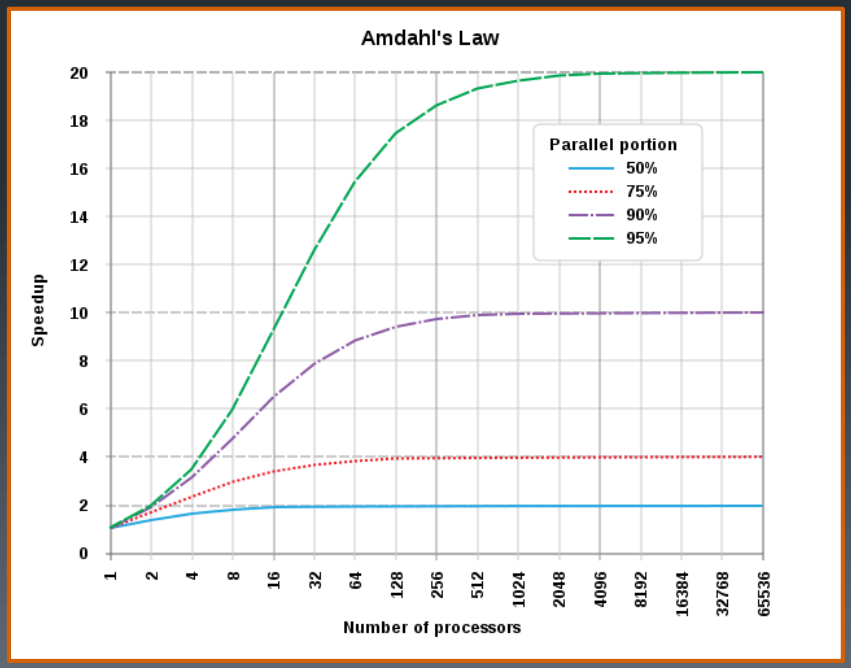
Objectives

- Understand Go's
  - Concurrency
  - Goroutines
  - Channels
  - Locking

Parallelism

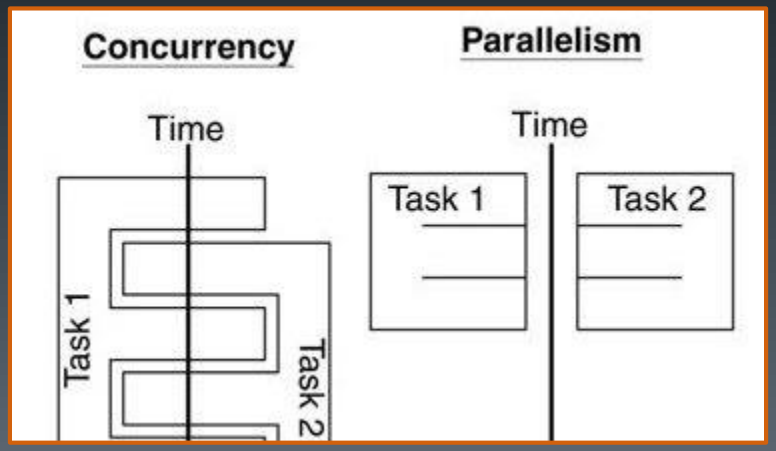
- Parallel computing implies simultaneous execution of processes/threads
  - Large problems can often be divided into smaller ones, which can then be solved at the same time, in parallel
- Types of parallel computing:
  - Bit Parallelism – Increasing cpu word size (the number of parallel bits processed) reduces the instructions the processor must execute to perform an operation
  - Instruction Parallelism – Increasing the number of instructions a computer can execute at the same time (multi core/processor)
  - Data Parallelism – Distributing data across multiple computers allowing them to process the data in parallel
  - Task Parallelism - Distributing tasks concurrently performed by processes or threads across different processors
- Physical constraints preventing frequency scaling have driven multiprocessor and multicore systems
  - Nearly all modern software systems have access to parallel computing (phones and many other embedded systems are typically multicore)
  - This places the burden of designing for task parallelism on developers who wish to take advantage of multiple cores/processors
- Parallel computers can be roughly classified according to the level at which the hardware supports parallelism
  - Multi-core and multi-processor computers having multiple processing elements within a single machine
  - Clusters, MPPs, and grids use multiple computers to work on the same task
- In some cases parallelism is transparent to the programmer
  - E.g. bit-level and instruction-level parallelism
- Parallel algorithms are more difficult to write than sequential ones
  - Concurrency introduces several new classes of software bugs
    - Race conditions are the most common
  - Communication and synchronization between different subtasks are some of the greatest obstacles to good parallel program performance
- Amdahl's law defines the upper bound of program parallel speed-up
  - $S_{latency}$ the theoretical speedup of the execution of the whole task
  - s the speedup of the part that benefits from improved resources
  - p the proportion of execution time that the part benefiting from improved resources originally occupied
  - E.g. Given
    - A program needs 20 hours on a single core to complete
    - A part of the program takes one hour to execute and cannot be parallelized
    - The remaining 19 hours (p = 0.95) of execution can be parallelized
    - At any core count the min execution time cannot be less than 1 hour
    - The theoretical speedup is limited to 20 times (1/(1 − p) = 20)

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$



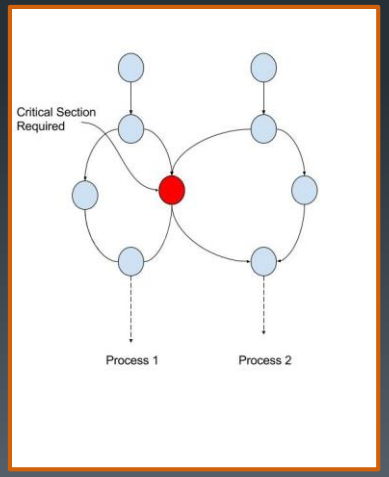
Concurrency

- Concurrency is the composition of independently executing things
  - -- Rob Pike
- Parallel computing is closely related to concurrent computing though the two are distinct:
  - It is possible to have parallelism without concurrency
    - Such as bit-level parallelism
  - It is possible to have concurrency without parallelism
    - Such as multitasking by time-sharing on a single-core CPU
- Concurrency is the decomposability a program into order-independent or partially-ordered components
  - This means that even if the concurrent units of a program are executed out-of-order or in partial order, the final outcome will remain the same
  - This allows for parallel execution of the concurrent units which can significantly improve overall speed of the execution in multi-processor and multi-core systems



Ordering

- Designing for concurrency involves decomposing a program, algorithm, or problem into order-independent or partially-ordered components
  - If the concurrent units of the program, algorithm, or problem are executed out-of-order or in partial order, the final outcome will remain the same
- The goal of concurrency is to improve overall speed of the execution in multi-processor and multi-core systems
- Mutual exclusion is a property of concurrency control
  - Purpose is to prevent race conditions
  - One thread of execution can not enter a critical section at the same time as another thread
  - A critical section is a piece of code where concurrent access can lead to undefined behavior
- Concurrency does not mean parallel
  - 1 CPU can run things concurrently, but not in parallel



Traditional Tools

C++

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void increment_count() {
  pthread_mutex_lock(&count_mutex);
  count = count + 1;
  pthread_mutex_unlock(&count_mutex);
}

long long get_count() {
  long long c;
  pthread_mutex_lock(&count_mutex);
  c = count;
  pthread_mutex_unlock(&count_mutex);
  return (c);
}
```

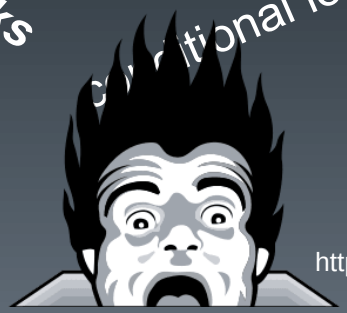
mutex lock

nested locking

locking hierarchies

deadlocks

conditional locking



Java

```
try {
  mutex.acquire();

  try {
    // do something
  } finally {
    mutex.release();
  }
} catch(InterruptedException ie) {
  // ...
}
```

https://blog.codinghorror.com/

Goroutines

- A goroutine is a function that is capable of running concurrently with other functions
- To create a goroutine we use the keyword go followed by a function invocation:
  - go f()
- All programs consist of at least one goroutine
  - The first goroutine is implicit and is the main function itself
  - Additional goroutines are created when arriving at go statements
- Normally when a function is invoked the program will execute all the statements in a function and then return to the next line following the invocation
  - goroutines return immediately and don't wait for the function to complete
- Goroutines are lightweight, programs can create thousands of them without extreme penalties

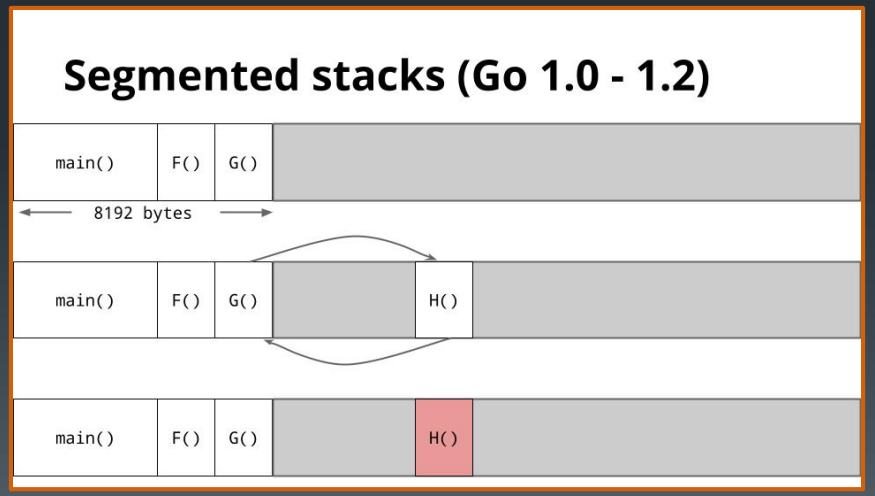
```
user@ubuntu:~/go/src/examples$ go run gorout.go
0 : 0
1 : 0
2 : 0
0 : 1
2 : 1
1 : 1
0 : 2
2 : 2
1 : 2

user@ubuntu:~/go/src/examples$
```

```go
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func f(n int) {
  for i := 0; i < 3; i++ {
    fmt.Println(n, ":", i)
    amt := time.Duration(rand.Intn(250))
    time.Sleep(time.Millisecond * amt)
  }
}

func main() {
  for i := 0; i < 3; i++ {
    go f(i)
  }
  var input string
  fmt.Scanln(&input)
}
```

Call Stacks

- goroutine:
  - Play on the related term coroutines
  - A function that executes concurrently with other goroutines in the same address space (process)
  - Lightweight
    - Little more than allocation of stack space
    - Stack memory is small at creation time, so it is cheap
      - 4096 byte initial stack
    - Uses heap to allocate larger stack if required
  - Multiplexes onto OS threads, if one blocks another goroutine will take over the OS thread and continue to run
    - Low level threading details are hidden
- To use:
  - Prefix your function call with “go”
  - Ex. “go myfunc()”
  - Like “async” in some languages
- Are implement as follows:
  - Small preamble is inserted before each function
  - Checks to see if function needs less than available memory (else call runtime-morestack)
  - Copies argument to new stack
  - Returns control to caller
  - When goroutine launched function finishes, process reverses
    - Return args are placed in caller stack
    - Memory is reclaimed
- While goroutines make it easy to run parallel code, ordering requires waiting/blocking, hence channels

```
var a string

func f() {
  print(a)
}

func hello() {
  a = "hello, world"
  go f()
}
```



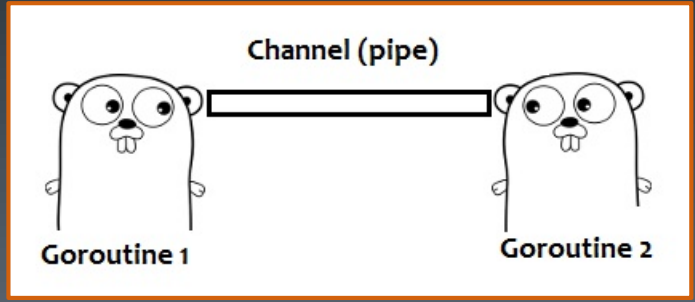
https://dave.cheney.net/wp-content/uploads/2014/06/Gocon-2014-43.jpg

Channels

- Channels are a typed conduit through which you can send and receive values with the channel operator: <-
  - Send to v channel c >> c <- v
  - Receive v from channel c >> v := <- c
- Channels are allocated via a call to make()
  - Example>> c := make(chan int)
- By default, sends and receives **block** until the other side is ready
  - Allows for synchronization without locks
- Channels can be unbuffered or buffered
  - Unbuffered combine exchange of data with synchronization
    - Aka. The sender blocks until the receiver has received the value
  - Buffered combine
    - Aka. The sender only blocks until the receiver has copied the value to the buffer, if buffer is full the sender waits
- Channels are the main method of communication between goroutines

```go
package main

var c = make(chan string)

func f() {
    a := "hello, world"
    c <- a
}

func main() {
    go f()
    x := <-c
    print(x)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/11 12:10:02 server.go:73: Using API v1
2017/04/11 12:10:02 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/11 12:10:02 debugger.go:414: continuing
hello, world
```



https://waimengmoan.gitbooks.io/go-applications/content/channel.jpg

Once

- Package “sync” provides a safe mechanism for initialization in presence of multiple goroutines
  - once.Do(f)
  - Function f is only called once

```
var a string
var once sync.Once

func setup() {
  a = "hello, world"
}

func doprint() {
  once.Do(setup)
  print(a)
}

func twoprint() {
  go doprint()
  go doprint()
}
```

Mutexes and Locks

- Package “sync” provides two lock data types
  - sync.Mutex
    - All locks are exclusive
  - sync.RWMutex
    - Write locks are exclusive
    - Read locks can be shared
- Provide mutual exclusion support
  - func (m *Mutex) Lock()
    - Lock locks m, blocks until the mutex is available
  - func (m *Mutex) Unlock()
    - Unlock unlocks m, run-time error if m is not locked

```
var l sync.Mutex

var a string

func f() {
  a = "hello, world"
  l.Unlock()
}

func main() {
  l.Lock()
  go f()
  l.Lock()
  print(a)
}
```

Conditions

- sync package type Cond
  - Cond implements a condition variable
  - A rendezvous point for goroutines waiting for or announcing the occurrence of an event
  - Each Cond has an associated Locker L (usually a *Mutex or *RWMutex) which must be held when changing the condition and when calling the Wait method
  - Can be created as part of other structures
  - Must not be copied after first use.
- func NewCond(l Locker) *Cond
  - Returns a new Cond with Locker l
- func (c *Cond) Broadcast()
  - Wakes all goroutines waiting on c
  - It is allowed but not required for the caller to hold c.L during the call
- func (c *Cond) Signal()
  - Wakes one goroutine waiting on c, if there is any
  - It is allowed but not required for the caller to hold c.L during the call
- func (c *Cond) Wait()
  - Atomically unlocks c.L and suspends execution of the calling goroutine
  - Wait locks c.L before returning
  - Wait cannot return unless awoken by Broadcast or Signal

```
14  func TestCondSignal(t *testing.T) {
15          var m Mutex
16          c := NewCond(&m)
17          n := 2
18          running := make(chan bool, n)
19          awake := make(chan bool, n)
20          for i := 0; i < n; i++ {
21                  go func() {
22                          m.Lock()
23                          running <- true
24                          c.Wait()
25                          awake <- true
26                          m.Unlock()
27                  }()
28          }
29          for i := 0; i < n; i++ {
30                  <-running // Wait for everyone to run.
31          }
32          for n > 0 {
33                  select {
34                  case <-awake:
35                          t.Fatal("goroutine not asleep")
36                  default:
37                  }
38                  m.Lock()
39                  c.Signal()
40                  m.Unlock()
41                  <-awake // Will deadlock if no goroutine wakes up
42                  select {
43                  case <-awake:
44                          t.Fatal("too many goroutines awake")
45                  default:
46                  }
47                  n--
48          }
49          c.Signal()
50  }
```

Pools

- sync package type Pool
  - type Pool struct {
    // Optional function to generate a value (may not be changed concurrently with calls to Get)
    New func() interface{}
    }
  - A Pool is a set of temporary objects that may be individually saved and retrieved
  - Any item stored in the Pool may be removed automatically
    - Safe for use by multiple goroutines simultaneously
- Pool's purpose is to cache allocated but unused items for later reuse, relieving pressure on the garbage collector
  - A thread-safe free lists
- Used to manage a group of temporary items silently shared among and potentially reused by concurrent independent clients of a package
  - Amortizes allocation overhead across many clients
  - e.g. The fmt package maintains a dynamically-sized store of temporary output buffers in a Pool
- Free lists for short-lived objects are not a suitable use for a Pool due to the Pool overhead
  - It is more efficient to have such objects implement their own free list
- A Pool must not be copied after first use
- func (p *Pool) Get() interface{}
  - Selects an arbitrary item from the Pool to return to the caller
  - Get may choose to ignore the pool and treat it as empty
    - If Get would otherwise return nil and p.New is non-nil, Get returns the result of calling p.New
- func (p *Pool) Put(x interface{})
  - Put adds x to the pool

Wait Groups

- The sync package provides the WaitGroup type
- WaitGroup
  - A type used to wait for a collection of goroutines to finish
  - The main goroutine calls Add to set the number of goroutines to wait for
  - Each of the goroutines runs and calls Done when finished
  - Must not be copied after first use.
- func (wg *WaitGroup) Add(delta int)
  - Adds delta, which may be negative, to the WaitGroup counter
  - If the counter becomes zero, all goroutines blocked on Wait are released
  - If the counter goes negative, Add panics
  - Calls with a positive delta that occur when the counter is zero must happen before a Wait
  - Calls with a negative delta, or calls with a positive delta that start when the counter is greater than zero, may happen at any time
    - Thus calls to Add should execute before the statement creating the goroutine or other event to be waited for
    - If a WaitGroup is reused to wait for several independent sets of events, new Add calls must happen after all previous Wait calls have returned
- func (wg *WaitGroup) Done()
  - Decrements the WaitGroup counter
- func (wg *WaitGroup) Wait()
  - Blocks until the WaitGroup counter is zero

```
var wg sync.WaitGroup
var urls = []string{
        "http://www.golang.org/",
        "http://www.google.com/",
        "http://www.somestupidname.com/",
}
for _, url := range urls {
        // Increment the WaitGroup counter.
        wg.Add(1)
        // Launch a goroutine to fetch the URL.
        go func(url string) {
                // Decrement the counter
                defer wg.Done()
                // Fetch the URL.
                http.Get(url)
        }(url)
}
// Wait for all HTTP fetches to complete.
wg.Wait()
```

Additional Concepts

- Pipelines
  - No formal definition
  - Informally a series of stages connected by channels
  - General flow:
    - Upstream provides input to goroutines via channel
    - goroutines process data
    - Downstream receives results from processing
  - Each stage can be many to many (m to n) channels
- Generator pattern
  - Return a channel
- Channel as reference to a service
- Multiplexing, and more.

```
c := boring("boring!") // Function returning a channel.
for i := 0; i < 5; i++ {
  fmt.Printf("You say: %q\n", <-c)
}

fmt.Println("You're boring; I'm leaving.")

func boring(msg string) <-chan string { // Returns receive-only channel of strings.
  c := make(chan string)
  go func() { // We launch the goroutine from inside the function.
    for i := 0; ; i++ {
       c <- fmt.Sprintf("%s %d", msg, i)
       time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
  }()
  return c // Return the channel to the caller.
}
```

```
func main() {
  joe := boring("Joe")
  ann := boring("Ann")

  for i := 0; i < 5; i++ {
    fmt.Println(<-joe)
    fmt.Println(<-ann)
  }

  fmt.Println("You're both boring; I'm leaving.")
}
```

https://talks.golang.org/2012/concurrency.slide

Summary

- Goroutines provide a simper alternative to heavy weight threads in other languages
- Channels provide a way to communicate between goroutines
- Don’t communicate by sharing memory, share memory to communicate

Lab: CSP and goroutines

- Work with goroutines and channels

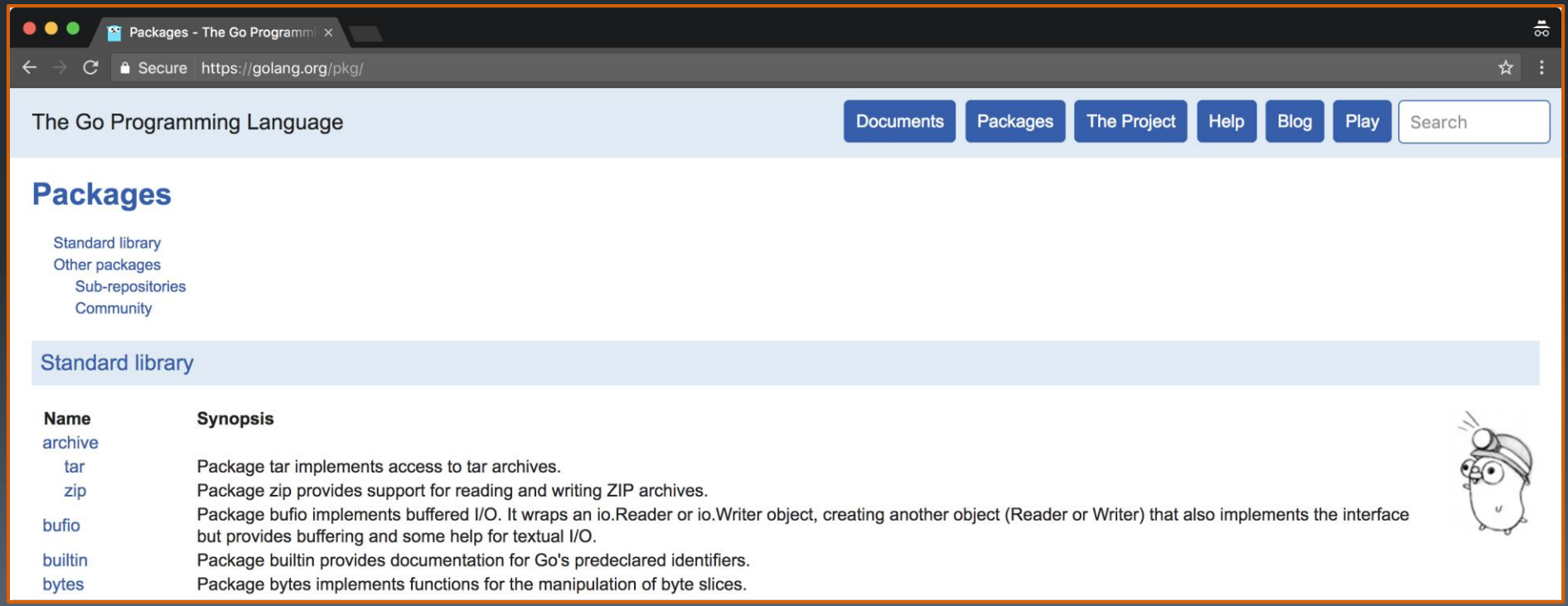
10: The Go standard library

Objectives

- Gain familiarity with the Go standard library
- Explore Sub Repositories
- Work with databases, regular expressions, networking features and other library packages

Standard Library Docs

- https://golang.org/pkg/

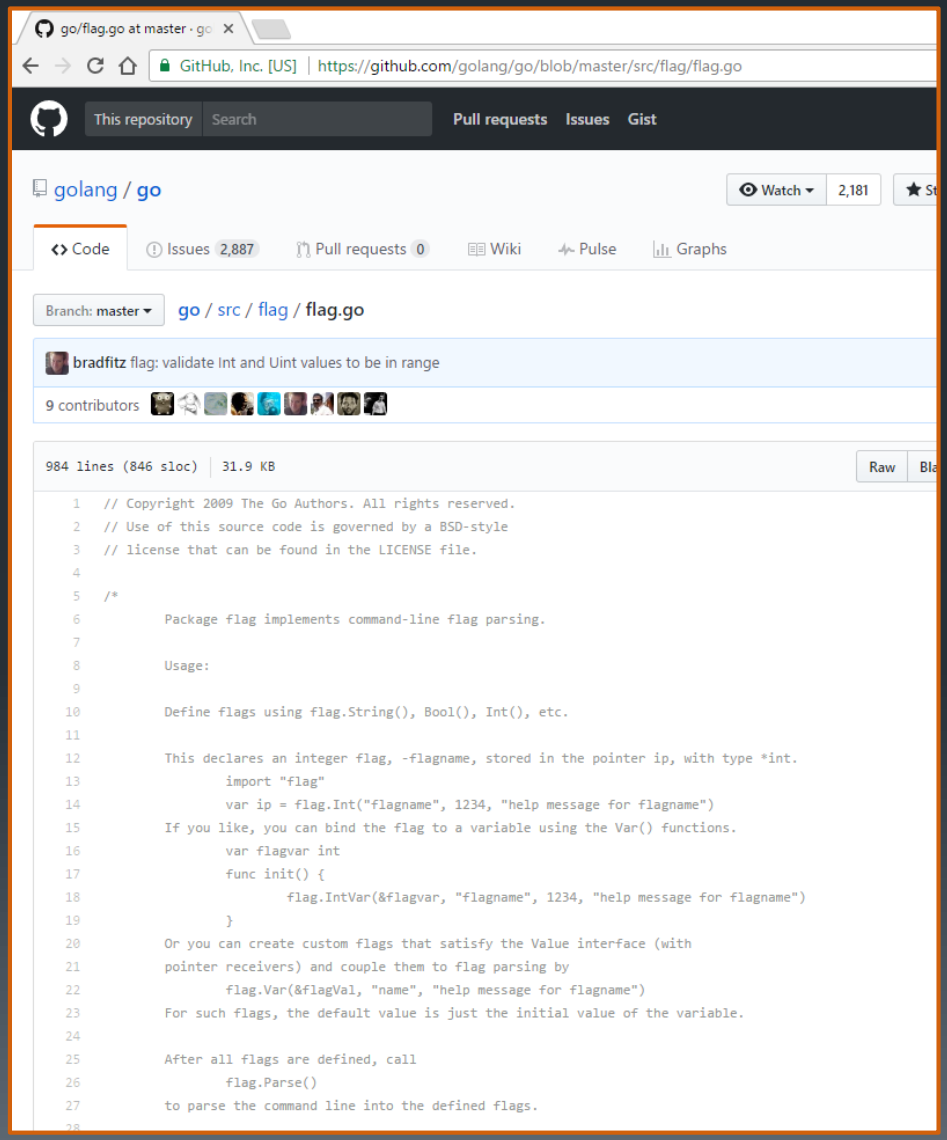


Standard Library Packages

- archive – implements tar and zip
- bufio – buffered i/o
- builtin – predeclared identifier documentation
- bytes - manipulation of byte slices
- compress - compression
- container – heap, lists, rings
- context – release related resources
  - https://blog.golang.org/context
- crypto – common cryptographic constants
- database – sql and database interfaces
- debug – debugging symbol and object access
- encoding – interfaces to convert byte to text
- errors – manipulate errors
- expvar – interface to expose public variables
- flag – command line parsing
- fmt – formatted i/o
- go – go compiler components
- hash – interfaces to hash functions
- html – escape html
- image – 2d image
- index – substring search
- io – i/o primitives
- log – simple logging package
- math – constants and functions
- mime – mime spec
- net – portable network interface
- os – platform independent os interface
- path - manipulate slash-separated paths
- plugin – loading and symbol resolution go plugins (linux only)
- reflect – run-time reflection
- regexp – regular expression search
- runtime – interface go runtime system
- sort – collection sorting
- strconv – to and from string
- strings – manipulate utf-8 encoded strings
- sync – synchronization primitives
- syscall – low-level os primitives
- testing – automated testing
- text – utf-8 tokenizer, writer, template engine
- time – measure and display time
- unicode – test unicode code points
- unsafe – bypass go type safety

Open Source

- The entire Go tool chain is open source
  - This includes the standard library
  - And it is all written in Go!
- Reading the standard library sources has many benefits
  - Teaches you idiomatic Go as well as good general coding patterns
    - The standard library is of very high code quality
  - Helps you better understand the nuances of the standard library and its behavior



Go Database Access

- Package sql
  - import "database/sql"
- Package sql provides a generic interface around SQL/SQLish databases
- Used in conjunction with a database driver
  - Driver list: https://golang.org/s/sqldrivers
- Drivers that do not support context cancelation will not return until after the query is completed

Drivers

Drivers for Go's sql package include:

- **Apache Phoenix/Avatica**: https://github.com/Boostport/avatica
- **ClickHouse**: https://github.com/kshvakov/clickhouse
- **Couchbase N1QL**: https://github.com/couchbase/go_n1ql
- **DB2**: https://bitbucket.org/phiggins/db2cli
- **Firebird SQL**: https://github.com/nakagami/firebirdsql
- **MS ADODB**: https://github.com/mattn/go-adodb
- **MS SQL Server** (pure go): https://github.com/denisenkom/go-mssqldb
- **MS SQL Server** (uses cgo): https://github.com/minus5/gofreetds
- **MySQL**: https://github.com/ziutek/mymysql `[*]`
- **MySQL**: https://github.com/go-sql-driver/mysql/ `[*]`
- **ODBC**: https://bitbucket.org/miquella/mgodbc
- **ODBC**: https://github.com/alexbrainman/odbc
- **Oracle**: https://github.com/mattn/go-oci8
- **Oracle**: https://github.com/rana/ora
- **QL**: http://godoc.org/github.com/cznic/ql/driver
- **Postgres** (pure Go): https://github.com/lib/pq `[*]`
- **Postgres** (uses cgo): https://github.com/jbarham/gopgsqldriver
- **Postgres** (pure Go): https://github.com/jackc/pgx
- **SAP HANA** (pure go): https://github.com/SAP/go-hdb
- **SQLite**: https://github.com/mattn/go-sqlite3 `[*]`
- **SQLite**: https://github.com/gwenn/gosqlite - Supports SQLite dynamic data typing
- **SQLite**: https://github.com/mxk/go-sqlite
- **Sybase SQL Anywhere**: https://github.com/a-palchikov/sqlago
- **Vitess**: https://godoc.org/github.com/youtube/vitess/go/vt/vitessdriver
- **YQL (Yahoo! Query Language)**: https://github.com/mattn/go-yql

Drivers marked with a `[*]` are both included in and pass the compatibility test suite at https://github.com/bradfitz/go-sql-test

Sample DB Client

- To import a package solely for its side effects (initialization), use the blank identifier as explicit package name:
  - import _ “github.com/…
- In the example we wish to initialize the mysql driver but we do not use it directly

```go
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:gorocks@/nums")
    if err != nil {
        panic(err.Error())
    }
    defer db.Close()

    //INSERT
    stmtIns, err := db.Prepare("INSERT INTO squarenum VALUES( ?, ? )")
    if err != nil {
        panic(err.Error())
    }
    defer stmtIns.Close()

    //SELECT
    stmtOut, err := db.Prepare("SELECT squarenum FROM squarenum WHERE number = ?")
    if err != nil {
        panic(err.Error())
    }
    defer stmtOut.Close()

    //Perform INSERTs
    for i := 0; i < 25; i++ {
        _, err = stmtIns.Exec(i, (i * i))
        if err != nil {
            panic(err.Error())
        }
    }

    var squareNum int

    //Perform SELECT
    err = stmtOut.QueryRow(13).Scan(&squareNum)
    if err != nil {
        panic(err.Error())
    }
    fmt.Printf("The square number of 13 is: %d\n", squareNum)

    err = stmtOut.QueryRow(1).Scan(&squareNum)
    if err != nil {
        panic(err.Error())
    }
    fmt.Printf("The square number of 1 is: %d\n", squareNum)
}
```

DB Walk Through

```
$ wget -qO- https://get.docker.com | sh

$ sudo docker run --name go-mysql --net=host -e MYSQL_ROOT_PASSWORD=gorocks -d mysql

$ sudo docker run -it --net=host --rm mysql sh -c 'exec mysql -h"127.0.0.1" -P"3306" -uroot -p"gorocks"'

mysql> create database nums;

mysql> use nums

mysql> create table squarenum (number INT, squarenum INT);

mysql> quit

$ go get github.com/go-sql-driver/mysql

$ go run db.go
The square number of 13 is: 169
The square number of 1 is: 1

$
```

The net package

- In addition to being the parent of the popular http package, the net package offers low level networking primitives directly
  - Listen to create a server
    - Accept to accept connections
  - Dial to connect to a server
  - Connection support:
    - io.Reader and io.Writer interfaces
    - Close to close the connection

```
user@ubuntu:~/go/src/examples$ go run server.go
Server listening on port  8088  ...
Message Received:Hi
Message Received:Bye
user@ubuntu:~/go/src/examples$
```

```
user@ubuntu: ~/go/src/examples
File Edit View Search Terminal Help
user@ubuntu:~/go/src/examples$ go run client.go
Text to send: Hi
Message from server:  HI

Text to send: Bye
Message from server:  BYE

Text to send: ^Csignal: interrupt
user@ubuntu:~/go/src/examples$
```

```go
package main

import "net"
import "fmt"
import "bufio"
import "os"

func main() {
        conn, _ := net.Dial("tcp", "127.0.0.1:8088")
        for {
                reader := bufio.NewReader(os.Stdin)
                fmt.Print("Text to send: ")
                text, _ := reader.ReadString('\n')
                fmt.Fprintf(conn, text + "\n")
                message, _ := bufio.NewReader(conn).ReadString('\n')
                fmt.Println("Message from server: ", message)
        }
}
```

```go
package main

import "net"
import "fmt"
import "bufio"
import "strings"

func main() {
        port := "8088"
        ln, _ := net.Listen("tcp", ":" + port)
        defer ln.Close()
        fmt.Println("Server listening on port ", port, " ...")

        conn, _ := ln.Accept()
        defer conn.Close()

        for {
                message, _ := bufio.NewReader(conn).ReadString('\n')
                if len(message) <= 0 {
                        return
                }
                fmt.Print("Message Received:", string(message))
                newmessage := strings.ToUpper(message)
                conn.Write([]byte(newmessage + "\n"))
        }
}
```

regex

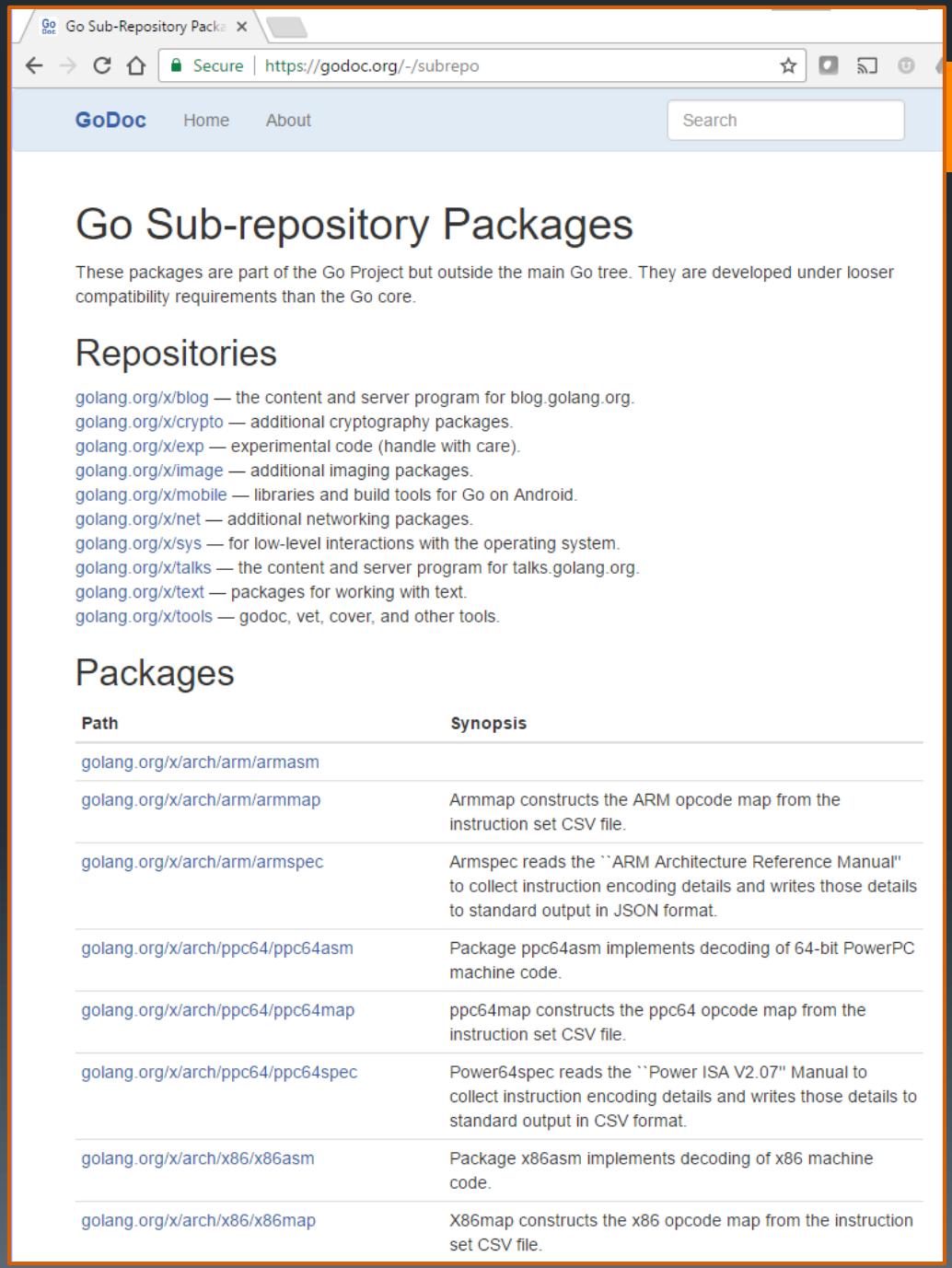
- Package regexp implements regular expression search
- Go regular expressions syntax is the same general syntax used by Perl, Python, and other languages
  - It is the syntax accepted by RE2 and described at https://golang.org/s/re2syntax
  - For an overview: go doc regexp/syntax
- regexp runs in linear time based on the size of the input
  - A property not guaranteed by most open source implementations of regular expressions
- All characters are UTF-8-encoded code points

```
user@ubuntu:~/go/src/examples$ go run reg.go
true
true
false
false
user@ubuntu:~/go/src/examples$
```

```go
package main

import "fmt"
import "regexp"

func main() {
        var validID = regexp.MustCompile(`^[a-z]+\[[0-9]+\]$`)

        fmt.Println(validID.MatchString("adam[23]"))
        fmt.Println(validID.MatchString("eve[7]"))
        fmt.Println(validID.MatchString("Job[48]"))
        fmt.Println(validID.MatchString("snakey"))
}
```

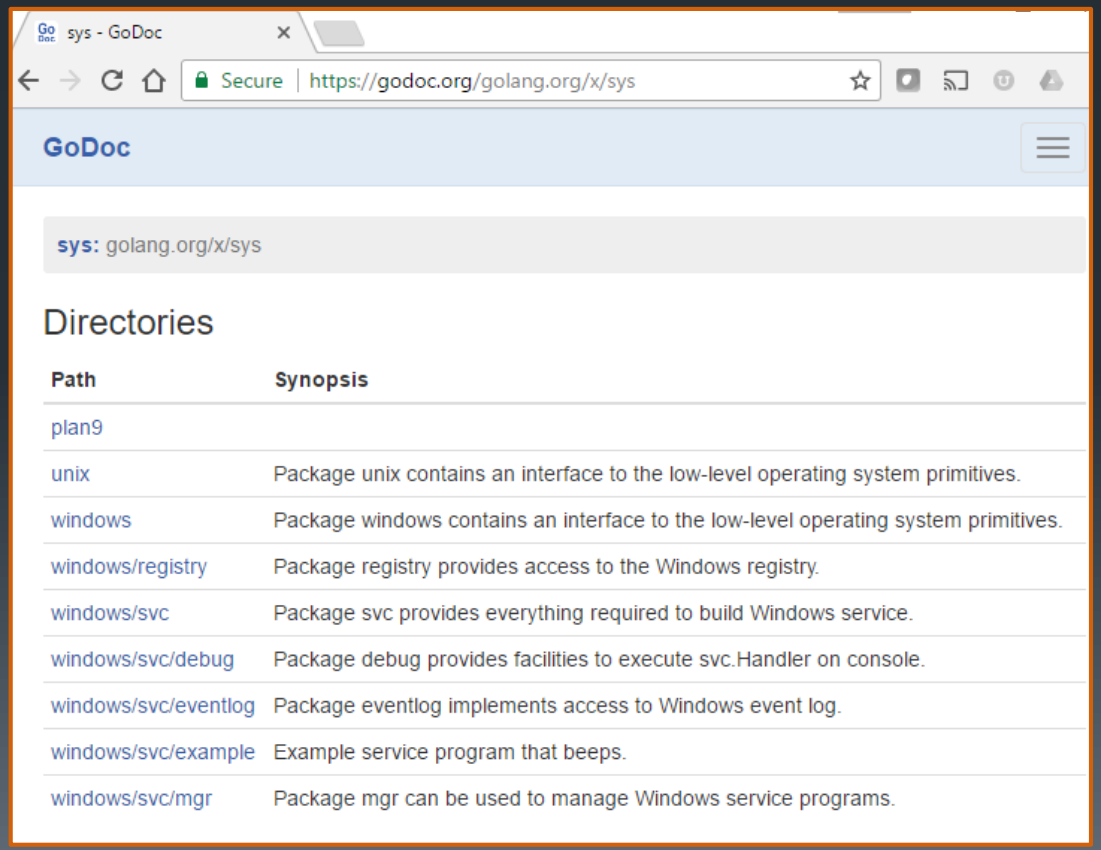
Go Sub-repository

- The golang.org/x/ repositories hold packages maintained by the Go team
  - Networking
  - internationalized text processing
  - mobile platforms
  - image manipulation
  - Cryptography
  - developer tools
- Packages not in the standard library because:
  - Still under development
  - Rarely needed by the majority of Go programmers
  - Conformance requirements are less strict



/x/sys

- An interface to low-level operating system primitives
  - Details vary depending on the underlying system
- Primary use is inside packages that provide a more portable interface to the system, such as "os", "time" and "net“
  - Use those packages if you can
- Details of the functions and data types in this package are found in the manuals for the appropriate operating system
- Returns err == nil to indicate success
  - Otherwise err is an operating system error describing the failure
  - On most systems, that error has type syscall.Errno
- golang.org/x/sys
  - The older syscall package should no longer be used



/x/sys/unix

- Low level networking and I/O functions are available here
  - E.g. epoll API

sys: golang.org/x/sys/unix Index | Files

package unix

```
import "golang.org/x/sys/unix"
```

Package unix contains an interface to the low-level operating system primitives. OS details vary depending on the underlying system, and by default, godoc will display OS-specific documentation for the current system. If you want godoc to display OS documentation for another system, set $GOOS and $GOARCH to the desired system. For example, if you want to view documentation for freebsd/arm on linux/amd64, set $GOOS to freebsd and $GOARCH to arm. The primary use of this package is inside other packages that provide a more portable interface to the system, such as "os", "time" and "net". Use those packages rather than this one if you can. For details of the functions and data types in this package consult the manuals for the appropriate operating system. These calls return err == nil to indicate success; otherwise err represents an operating system error describing the failure and holds a value of type syscall.Errno.

Index

Constants
Variables
func Accept(fd int) (nfd int, sa Sockaddr, err error)
func Accept4(fd int, flags int) (nfd int, sa Sockaddr, err error)
func Access(path string, mode uint32) (err error)
func Acct(path string) (err error)
func Adjtimex(buf *Timex) (state int, err error)
func Bind(fd int, sa Sockaddr) (err error)
func BindToDevice(fd int, device string) (err error)
func BytePtrFromString(s string) (*byte, error)

```
package main

import "golang.org/x/sys/unix"
import "fmt"

func main() {
        fmt.Println("System PageSize: ", unix.Getpagesize())
}
```

```
user@ubuntu:~/go/src/examples$ go run sys.go
System PageSize:  4096
user@ubuntu:~/go/src/examples$
```

spec) (err error)

func CmsgSpace(datalen int) int
func Connect(fd int, sa Sockaddr) (err error)
func CopyFileRange(rfd int, roff *int64, wfd int, woff *int64, len int, flags int) (n int, err error)
func Creat(path string, mode uint32) (fd int, err error)

Summary

- Go provides an extensive standard library
- Sub Repositories extend the available Go packages even further
- The Go standard library is written in Go
  - Making it a good source for learning go style an idiomatic coding patterns

Lab: Standard Library

- Standard Library

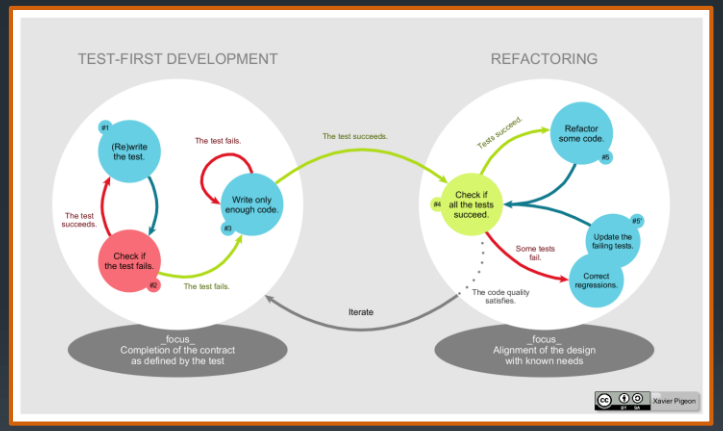
11: Testing

Objectives

- TDD and BDD
  - Concepts
  - Examples
- “go test”
  - usage
- Package “testing”
  - Required syntax
  - Examples

Test Driven Development (TDD)

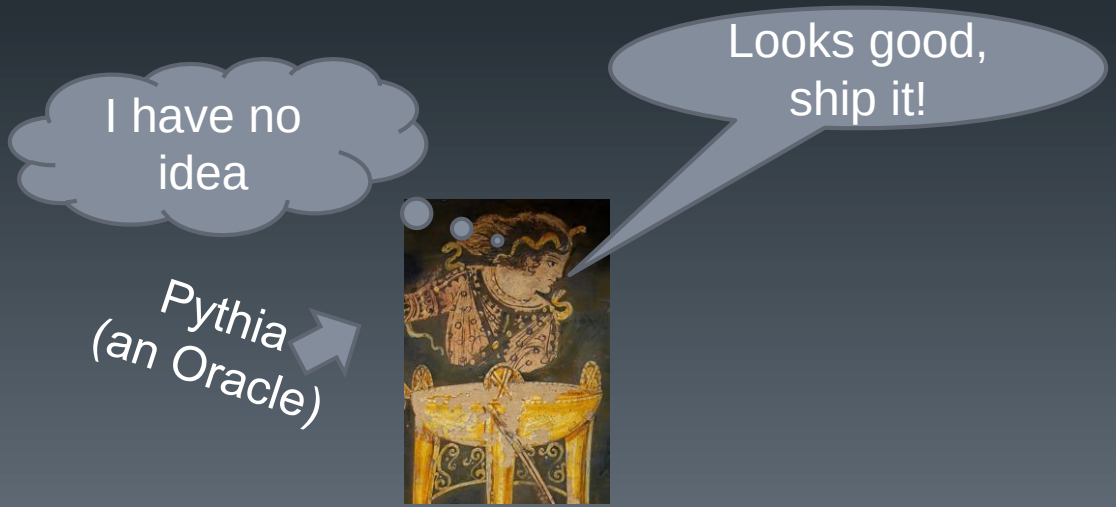
- TDD is a software development process that relies on very short development cycles
  - Tests are written first
  - Then code is added to make the test pass
- TDD appeared 1999 related to extreme programming (by Kent Beck)
- TDD lifecycle
  - Add a test
  - Run all test and see if the new test fails
  - Write the code
  - Run tests to see if new tests pass
  - Refactor code
  - Repeat
- Tests are generally small and self documenting
  - Called “unit tests”
- TDD is not the same as acceptance test driven development (ATDD)
  - ATDD is where customer level functionality is tested, often not automated
  - ATDD provides guidance on what should be an automated TDD test



TDD Practices

- Individual Tests
  - Move setup/teardown into a service (reduce duplication of code)
  - Keep focus of each test **oracle** (don't have single oracle across multiple test domains – aka. Source of truth)
  - Allow for skew in time based tests (+/-)
  - Treat test code like production code

- Anti-patterns (Don't do)
  - Depend on previous test results for current test
  - Test order should matters
  - Not testing time for time dependent code
  - All in one oracle
  - Testing implementation versus interface
  - Leaving slow tests



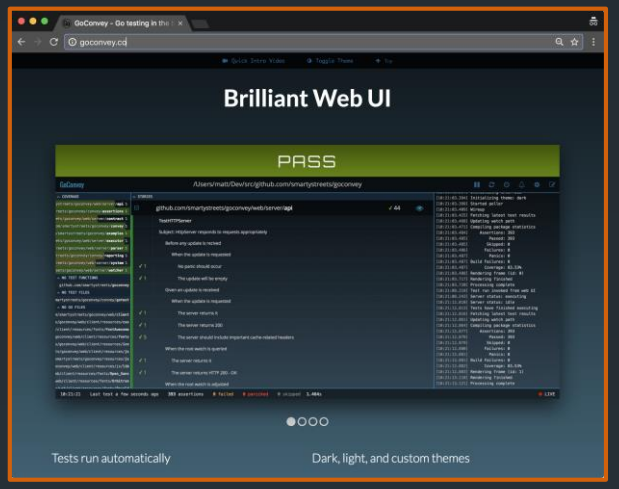

```
func TestLoop(t *testing.T) {
  cases := []struct {
    in int
    want int
  } {
    {-1,1},
    {3,6},
    {0,0},
 }

  for _, c := range cases {
    got := loop(c.in)
    if got != c.want {
      t.Errorf("loop(%d) == %d,
        want %d", c.in, got, c.want)
    }
  }
}
```

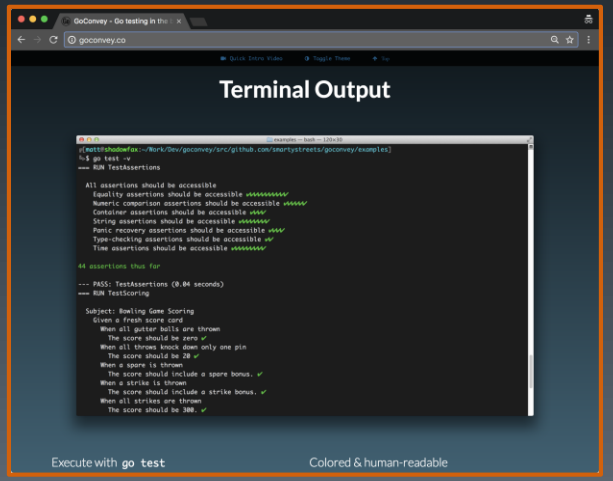
Behavior Driven Development (BDD)

- BDD combines TDD and ATDD
  - TDD – write tests first
  - ATDD – focus on behavior over implementation
- BDD emerges from TDD
- Allows for management and development to collaborate on what the behavior is for a given program and how to test it
- BDD is mostly facilitated via a domain specific language (DSL)
- BDD uses a combination of unit test (TDD) with acceptance test (ATDD)
- A TDD unit test is often non-specific, BDD focuses the test on desired behavior (aka business goal)
- BDD is considered to be at odds with Go design. The addition of a DSL is considered unnecessary complexity.
- BDD support for Go is provided by 3rd parties
  - One popular BDD package for Go is goconvey
  - https://github.com/smartystreets/goconvey
  - http://goconvey.co/

Example goconvey


```
package package_name

import (
  "testing"
  . "github.com/smartystreets/goconvey/convey"
)

func TestSpec(t *testing.T) {
  // Only pass t into top-level Convey calls
  Convey("Given some integer with a starting value", t, func() {
    x := 1

    Convey("When the integer is incremented", func() {
      x++

      Convey("The value should be greater by one", func() {
        So(x, ShouldEqual, 2)
      })
    })
  })
}
```





Package “testing”

- Package testing provides support for automated testing of Go packages
- Used in concert with the “go test” command
- Provides three primary function types
  - Test – code correctness
  - Benchmark – code timing
  - Example – code examples
- Additional functionality include:
  - Subtests and Sub-benchmarks
  - Parallelism
- “go test” looks for files that have a suffix “_test.go” and belong to the package of code being tested.

Using Test

- To create an automated test (one ran by “go test”)
  - Create *name*_test.go (*name* is your program name)
  - Define the test
    - Setup
    - Execute
    - Teardown
  - Run the test “go test”
    - Fail
    - Create *name*.go (name is your program name)
      - Implement the code
    - Rerun
    - Fix
    - Repeat
- “go test” keys off of “TestName”
- “*testing.T” is the parameter used by the test driver (to control the test state)
  - Report failures through the instance of ”t *testing.T”

```
user@ubuntu:~/go/src/gonuts$ go test -run=TestLoop
PASS
ok  gonuts 0.002s
user@ubuntu:~/go/src/gonuts$
```

loop.go

```
user@ubuntu:~/go/src/gonuts$ cat loop.go
package main


func loop(iter int) int {
  i := 0
  total := 0
  for i <= iter {
    total += i
    i++
  }
  return total
}
user@ubuntu:~/go/src/gonuts$
```

loop_test.go

```
package main

import "testing"

func TestLoop(t *testing.T) {
  cases := []struct {
    in int
    want int
  } {
    {-1,1},
    {3,6},
    {0,0},
 }

  for _, c := range cases {
    got := loop(c.in)
    if got != c.want {
      t.Errorf("loop(%d) == %d,
        want %d", c.in, got, c.want)
    }
  }
}
```

Using Benchmark

- Place benchmark code in *NAME*_test.go (can be same as TestFunc)
- Instead of Test prefix, use Benchmark prefix
- Like Test, there is object provided of type "*testing.B"
  - In the example "b.N" is calculated by "go test" to find a decent high mark on the number of times to execute the test
- To run, we need to supply the "-bench=*REGEX*" flag
  - "." means run anything with "Benchmark" in function name prefix

```
func BenchmarkLoop(b *testing.B) {
  for i := 0; i < b.N; i++ {
    loop(1000)
  }
}
```

```
user@ubuntu:~/go/src/gonuts$ go test -bench=BenchmarkLoop
BenchmarkLoop-2    5000000       307 ns/op
PASS
ok  gonuts 1.891s
user@ubuntu:~/go/src/gonuts$
```

Using Example

- Similar to Test and Benchmark, prefix example code (used in documentation) with Example
- When generating documentation via “godoc”
  - If no return value, it will compile
  - If return value, it will compile and execute
    - If there is output, using “// Output: 6” will compare to the actual result
- To assist godoc with doc generation we add a suffix to Example:
  - Nothing (top level package)
  - F function
  - T type
  - M method
- When used with “go test” it will test our example code!

```
func ExampleFLoop() {
  fmt.Println(loop(3))
  // Output: 6
}
```

Detecting Test Runs

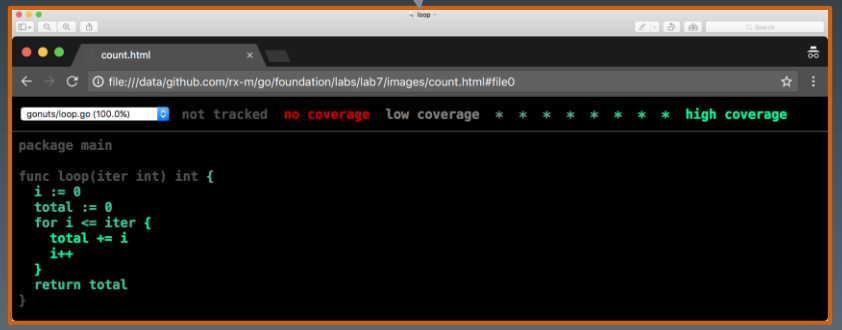
- The testing package modifies the global environment when loaded
  - Registering command-line flags etc.
- You can determine if you are running under test by checking for the “test.v” command line flag

```
func init() {
    if flag.Lookup("test.v") == nil {
        fmt.Println("normal run")
    } else {
        fmt.Println("run under go test")
    }
}
```

Coverage

- Code coverage is a measure to describe the degree of source code executed as part of a test
- To run “go test -coverprofile=output.txt”
- If you supply the “-covermode=*mode*” to change behavior
  - set – does this statement run (bool check)
  - count – how many times does this statement run (int check)
  - atomic – count for multi-threaded applications (int check, expensive)

```
user@ubuntu:~/go/src/gonuts$ go test -coverprofile=cover.out
PASScoverage: 75.0% of statements
ok          gonuts       0.002s
user@ubuntu:~/go/src/gonuts$

user@ubuntu:~/go/src/gonuts$ go tool cover -html=count.out -o count.html
```



Summary

- Go provides a basic TDD package called “testing”
- Exclusion of BDD DSL was by design
  - Core Go engineers believe to much TDD abstraction or BSD DSL hides valuable details
- Many additional flags and methods are available
  - Simple at first (by design)
  - Ability to add your own testing practices on top

Lab: Testing

- Develop basic tests and compare performance against algorithms that produce the same output but differ in implementation.

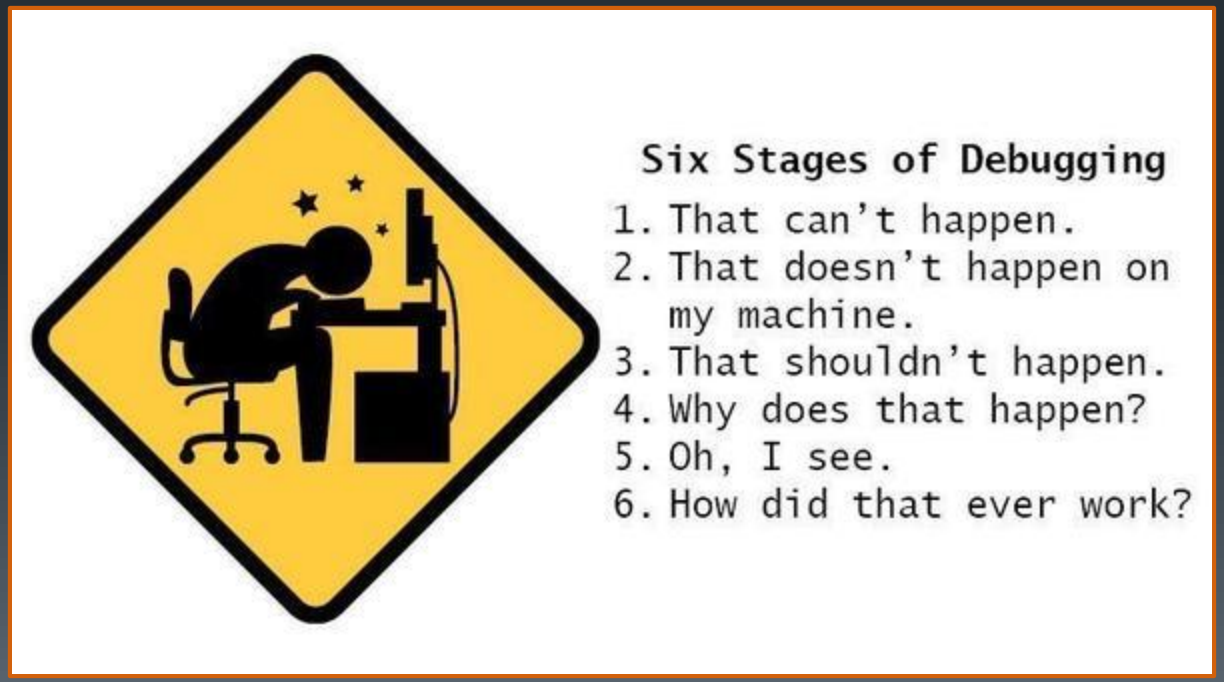
Go Debugging

Objectives

- Explain the challenges of debugging Go programs
- List some Go Debugging tools
- Describe the use of Delve

Debugging

- Debugging is the process of finding and resolving of defects that prevent correct operation of computer software or a system
  --wikipedia
- Debugging tactics can involve:
  - Interactive debugging
    - In Go: gdb or dlv
  - Control flow analysis:
    - Trace
  - Unit testing
    - Go Test
  - Integration testing
    - Go Test
  - Log file analysis
    - Package “log”
  - Monitoring
    - Various external tools
  - Memory dumps
    - gdb or dlv
  - Profiling
    - Pprof



GDB

- The GNU Project debugger
- Allows you to see what is going on `inside' another program while it executes
  - -- or what another program was doing at the moment it crashed
- GDB can do four main kinds of things:
  - Start your program, specifying anything that might affect its behavior.
  - Make your program stop on specified conditions.
  - Examine what has happened, when your program has stopped.
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
- The program being debugged can be written in Ada, C, C++, Objective-C, Pascal, Go and many other languages
- Those programs might be executing on the same machine as GDB (native) or on another machine (remote)
  - GDB can run on most popular UNIX and Microsoft Windows variants
- GDB does not understand Go programs well
  - Go stack management, threading, and the Go runtime differ from the execution model GDB expects
    - This is true even when the program is compiled with gccgo
  - GDB can be useful in some situations but it is not a reliable debugger for Go programs
    - It falls particularly short in heavily concurrent debugging because of its lack of recognition of goroutines
- A more Go-centric debugging architecture was required

Delve

- Delve (dlv) is a debugger for the Go programming language
- The goal of the project is to provide a simple, full featured debugging tool for Go
- Delve is easy to invoke and easy to use
- Delve is a “go get” able Go package
- Can be executed at the command line using dlv or through IDEs
  - Visual Studio Code, Eclipse Go and Gogland all use dlv under the covers

Delve Command Line


```
user@ubuntu:~/go/src/lab-debug$ dlv -help
Error: unknown shorthand flag: 'e' in -elp
Usage:
  dlv [command]

Available Commands:
  attach      Attach to running process and begin debugging.
  connect     Connect to a headless debug server.
  core        Examine a core dump.
  debug       Compile and begin debugging main package in current directory, or the package s
  exec        Execute a precompiled binary, and begin a debug session.
  help        Help about any command
  run         Deprecated command. Use 'debug' instead.
  test        Compile test binary and begin debugging program.
  trace       Compile and begin tracing program.
  version     Prints version.

Flags:
      --accept-multiclient   Allows a headless server to accept multiple client connections.
 to coordinate.
      --api-version int      Selects API version when headless. (default 1)
      --backend string       Backend selection:
        default         Uses lldb on macOS, native everywhere else.
        native          Native backend.
        lldb            Uses lldb-server or debugserver.
        rr              Uses mozilla rr (https://github.com/mozilla/rr).
 (default "default")
      --build-flags string   Build flags, to be passed to the compiler.
      --headless             Run debug server only, in headless mode.
      --init string          Init file, executed by the terminal client.
  -l, --listen string        Debugging server listen address. (default "localhost:0")
      --log                  Enable debugging server logging.
      --wd string            Working directory for running the program. (default ".")

Use "dlv [command] --help" for more information about a command.
```

Delve Session

```
user@ubuntu:~/go/src/lab-debug$ dlv debug more.go
Type 'help' for list of commands.
(dlv) break main.main
Breakpoint 1 set at 0x47d803 for main.main() ./more.go:14
(dlv) next
> _rt0_amd64_linux() /usr/local/go/src/runtime/rt0_linux_amd64.s:9 (PC: 0x44e365)
     4:
     5: #include "textflag.h"
     6:
     7: TEXT _rt0_amd64_linux(SB),NOSPLIT,$-8
     8:         LEAQ    8(SP), SI // argv
=>   9:         MOVQ    0(SP), DI // argc
    10:         MOVQ    $main(SB), AX
    11:         JMP     AX
    12:
    13: // When building with -buildmode=c-shared, this symbol is called when the shared
    14: // library is loaded.
(dlv) continue
> main.main() ./more.go:14 (hits goroutine(1):1 total:1) (PC: 0x47d803)
     9:         fmt.Printf("goroutine id %d\n", i)
    10:         fmt.Printf("goroutine id %d\n", i)
    11:         wg.Done()
    12: }
    13:
=>  14: func main() {
    15:         var wg sync.WaitGroup
    16:         workers := 10
    17:
    18:         wg.Add(workers)
    19:         for i := 0; i < workers; i++ {
(dlv) next
> main.main() ./more.go:15 (PC: 0x47d811)
    10:         fmt.Printf("goroutine id %d\n", i)
    11:         wg.Done()
    12: }
    13:
    14: func main() {
=>  15:         var wg sync.WaitGroup
    16:         workers := 10
    17:
    18:         wg.Add(workers)
    19:         for i := 0; i < workers; i++ {
```

next – steps over the next line
list – lists code at the current line
continue – runs to the next breakpoint
<enter> - reruns the previous command

Delve Commands


```
(dlv) h
The following commands are available:
    args ------------------------ Print function arguments.
    break (alias: b) ------------ Sets a breakpoint.
    breakpoints (alias: bp) ----- Print out info for active breakpoints.
    clear ----------------------- Deletes breakpoint.
    clearall -------------------- Deletes multiple breakpoints.
    condition (alias: cond) ----- Set breakpoint condition.
    continue (alias: c) --------- Run until breakpoint or program termination.
    disassemble (alias: disass) - Disassembler.
    exit (alias: quit | q) ------ Exit the debugger.
    frame ----------------------- Executes command on a different frame.
    funcs ----------------------- Print list of functions.
    goroutine ------------------- Shows or changes current goroutine
    goroutines ------------------ List program goroutines.
    help (alias: h) ------------- Prints the help message.
    list (alias: ls) ------------ Show source code.
    locals ---------------------- Print local variables.
    next (alias: n) ------------- Step over to next source line.
    on -------------------------- Executes a command when a breakpoint is hit.
    print (alias: p) ------------ Evaluate an expression.
    regs ------------------------ Print contents of CPU registers.
    restart (alias: r) ---------- Restart process.
    set ------------------------- Changes the value of a variable.
    source ---------------------- Executes a file containing a list of delve commands
    sources --------------------- Print list of source files.
    stack (alias: bt) ----------- Print stack trace.
    step (alias: s) ------------- Single step through program.
    step-instruction (alias: si)  Single step a single cpu instruction.
    stepout --------------------- Step out of the current function.
    thread (alias: tr) ---------- Switch to the specified thread.
    threads --------------------- Print out info for every traced thread.
    trace (alias: t) ------------ Set tracepoint.
    types ----------------------- Print list of types
    vars ------------------------ Print package variables.
Type help followed by a command for full documentation.
```

Navigating

- s – step into function
- n – step over
- stepout – step out of function
- si – single step a machine instruction

```
(dlv) s
> fmt.Println() /usr/local/go/src/fmt/print.go:256 (PC: 0x47489f)
   251: }
   252:
   253: // Println formats using the default formats for its operan
   254: // Spaces are always added between operands and a newline i
   255: // It returns the number of bytes written and any write err
=> 256: func Println(a ...interface{}) (n int, err error) {
   257:         return Fprintln(os.Stdout, a...)
   258: }
   259:
   260: // Sprintln formats using the default formats for its opera
   261: // Spaces are always added between operands and a newline i
(dlv) n
> fmt.Println() /usr/local/go/src/fmt/print.go:257 (PC: 0x4748ad)
   252:
   253: // Println formats using the default formats for its operan
   254: // Spaces are always added between operands and a newline i
   255: // It returns the number of bytes written and any write err
   256: func Println(a ...interface{}) (n int, err error) {
=> 257:         return Fprintln(os.Stdout, a...)
   258: }
   259:
   260: // Sprintln formats using the default formats for its opera
   261: // Spaces are always added between operands and a newline i
   262: func Sprintln(a ...interface{}) string {
(dlv) stepout
3
> main.main() ./sl.go:10 (PC: 0x47b8da)
     5: func main() {
     6:         var x [2]int
     7:         s1 := x[0:2]
     8:         s2 := x[0:2]
     9:         fmt.Println(x[1])
=>  10:         fmt.Println(s1[1])
    11:         fmt.Println(s2[1])
    12:         s1[1] = 6
    13:         fmt.Println(x[1])
    14:         fmt.Println(s1[1])
    15:         fmt.Println(s2[1])
(dlv)
```

Variables

- The print command can display the value of variables
- Print also understands many standard Go expressions
  - print x
  - print x < y
  - print &x
  - print x[4]
- The set command allows you to modify variables
  - set x = 5

```
(dlv)
> main.main() ./slice.go:9 (PC: 0x47b826)
     4:
     5: func main() {
     6:         var x [2]int
     7:         s1 := x[0:2]
     8:         s2 := x[0:2]
=>   9:         fmt.Println(x[1])
    10:         fmt.Println(s1[1])
    11:         fmt.Println(s2[1])
    12:         s1[1] = 6
    13:         fmt.Println(x[1])
    14:         fmt.Println(s1[1])
(dlv) print &x
(*[2]int)(0xc42003bc38)
(dlv) print &s1[0]
(*int)(0xc42003bc38)
(dlv) print &s2[0]
(*int)(0xc42003bc38)
(dlv) next
0
```

```
(dlv) set x[1] = 3
(dlv) print x
[2]int [0,3]
(dlv)
```

Breakpoints

- The break command sets breakpoints
  - On functions
  - On lines in a file
  - On lines offset from a function
- The bp command displays breakpoints
- The clear command clears a breakpoint
- The clearall command clears all breakpoints

```
user@ubuntu:~/go/src/slice$ dlv debug sl.go
Type 'help' for list of commands.
(dlv) b main.main
Breakpoint 1 set at 0x47b7ab for main.main() ./sl.go:5
(dlv) b sl.go:12
Breakpoint 2 set at 0x47ba76 for main.main() ./sl.go:12
(dlv) b main.main:+3
Breakpoint 3 set at 0x47b7fd for main.main() ./sl.go:8
(dlv)
```

```
(dlv) bp
Breakpoint unrecovered-panic at 0x4262b0 for runtime.startpa
        print runtime.curg._panic.arg
Breakpoint 1 at 0x47b7ab for main.main() ./sl.go:5 (1)
Breakpoint 2 at 0x47ba76 for main.main() ./sl.go:12 (0)
Breakpoint 3 at 0x47b7fd for main.main() ./sl.go:8 (0)
(dlv) clear 2
Breakpoint 2 cleared at 0x47ba76 for main.main() ./sl.go:12
(dlv) clearall
Breakpoint 1 cleared at 0x47b7ab for main.main() ./sl.go:5
Breakpoint 3 cleared at 0x47b7fd for main.main() ./sl.go:8
(dlv)
```

Debugging concurrent applications

- Delve allows you to switch and manage
  - Threads
    - threads – lists available threads
    - thread – switches to the specified thread
  - GoRoutines
    - goroutines – lists available goroutines
    - goroutine – switches to the specified goroutine

```
(dlv) goroutines
[14 goroutines]
  Goroutine 1 - User: /usr/local/go/src/runtime/sema.go:47 sync.runtime_Semacquire (0x435154)
  Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 17 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 18 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 19 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 20 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 21 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 22 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 23 - User: ./more.go:9 main.dostuff (0x47d60f) (thread 62728)
  Goroutine 24 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 25 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 26 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 27 - User: ./more.go:8 main.dostuff (0x47d5e0)
* Goroutine 28 - User: ./more.go:9 main.dostuff (0x47d60f) (thread 62720)
(dlv) goroutine 19
Switched from 28 to 19 (thread 62720)
(dlv) threads
* Thread 62720 at 0x47d60f ./more.go:9 main.dostuff
  Thread 62726 at 0x44e491 /usr/local/go/src/runtime/sys_linux_amd64.s:98 runtime.usleep
  Thread 62727 at 0x44e933 /usr/local/go/src/runtime/sys_linux_amd64.s:426 runtime.futex
  Thread 62728 at 0x47d60f ./more.go:9 main.dostuff
(dlv) goroutines
[14 goroutines]
  Goroutine 1 - User: /usr/local/go/src/runtime/sema.go:47 sync.runtime_Semacquire (0x435154)
  Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 17 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
  Goroutine 18 - User: /usr/local/go/src/runtime/proc.go:272 runtime.gopark (0x427f4a)
* Goroutine 19 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 20 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 21 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 22 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 23 - User: ./more.go:9 main.dostuff (0x47d60f) (thread 62728)
  Goroutine 24 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 25 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 26 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 27 - User: ./more.go:8 main.dostuff (0x47d5e0)
  Goroutine 28 - User: ./more.go:9 main.dostuff (0x47d60f) (thread 62720)
```

Other Commands

- You can display source files associated with the executable
  - sources
- You can display the call stack
  - stack
- You can display loaded types
  - types <optional regex>
- You can display vars defined
  - vars <optional regex>
- You can display functions defined
  - funcs <optional regex>

```
(dlv) sources
/home/user/go/src/slice/sl.go
/usr/local/go/src/errors/errors.go
/usr/local/go/src/fmt/doc.go
/usr/local/go/src/fmt/format.go
/usr/local/go/src/fmt/print.go
/usr/local/go/src/fmt/scan.go
/usr/local/go/src/io/io.go
/usr/local/go/src/io/pipe.go
/usr/local/go/src/math/abs.go
/usr/local/go/src/math/floor_amd64.s
/usr/local/go/src/math/floor_asm.go
/usr/local/go/src/math/pow10.go
/usr/local/go/src/math/unsafe.go
/usr/local/go/src/os/dir.go
/usr/local/go/src/os/error.go
/usr/local/go/src/os/exec_unix.go
/usr/local/go/src/os/executable_procfs.go
```

```
(dlv) stack
0  0x000000000047b8da in main.main
   at ./sl.go:10
1  0x0000000000427b0a in runtime.main
   at /usr/local/go/src/runtime/proc.go:185
2  0x000000000044d4a1 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:2197
(dlv)
```

```
(dlv) funcs strconv.App
strconv.AppendFloat
strconv.AppendQuote
strconv.AppendQuoteRune
strconv.AppendQuoteRuneTo
strconv.AppendQuoteToASCI
(dlv)
```

```
(dlv) vars ^main
main.initdone· = 2
(dlv) vars ^math
math.pow10tab = [70]float64 [...]
math.initdone· = 2
math.useSSE4 = true
(dlv)
```

```
(dlv) types ^sync\.(RWM|M)
sync.Mutex
sync.RWMutex
(dlv)
```

Remote Debugging

- Delve supports remote debugging
- Launch the target with dlv using the --headless switch
  - $ dlv --headless -l “192.168.0.4:9090” hello.go
- Other useful switches:
  - -l, --listen string Debugging server listen address
    - default "localhost:0"
  - --log Enable debugging server logging
- To attach to the remote debugger:
  - $ dlv connect 192.168.0.4:9090

Remote debugging in VSCode

- Each IDE has its own configuration approach
- VSCode uses a launch.json file to define launch configuration
- The example to the right shows a VSCode launch.json for use with a docker container configured for remote debugging

```
{
    "version": "0.2.0",
    "debugServer": 4711,
    "configurations": [
        {
            "name": "Launch",
            "type": "go",
            "request": "launch",
            "mode": "debug",
            "program": "${workspaceRoot}",
            "env": {},
            "args": []
        },
        {
            // To remote debug in Docker, run the following before debugging:
            // # docker build -t webapp-go .
            // # docker run -d --name webapp-go --privileged -p 8080:8080 -p 2345:2345 webapp-go
            // And then each time you want to restart debugging:
            // # docker restart
            "name": "Remote debug in Docker",
            "type": "go",
            "request": "launch",
            "mode": "remote",
            "program": "${workspaceRoot}",
            "env": {},
            "args": [],
            "remotePath": "/go/src/app",
            "port": 2345, // Port
            "host": "192.168.99.100" // Docker IP
        }
    ]
}
```

Summary

- There are many tools that can help with debugging Go programs
- Interactive debuggers are particularly useful
  - Solving problems
  - Also helping developers understand running programs and thus doing a better job of coding
- GDB is a stop gap debugger only
- Delve is the most useful Go debugger today

Lab

- Debugging