Learn How To Code
Google's Go (golang) Programming Language

Welcome

1. **welcome**

Welcome to ...

2. **credentials**

My name is Todd McLeod and ...

3. **benefits**

I am excited to show you ...

4. **curriculum**

The curriculum we will be studying ...

5. **audience**

For those new to programming … For those already with experience ...

6. **call to action**

video: 001

Course overview

Why Go?

- Go is the top paying language in America
- Credentials of Go
    - Google
    - **Rob Pike**
        - Unix, UTF-8
    - **Robert Griesemer**
        - Studied under the creator of Pascal
    - **Ken Thompson**
        - solely responsible for designing and implementing the original Unix operating system
        - invented the B programming language, the direct predecessor to the C programming language.
        - helped invent the C programming language
- Why Go
    - **efficient compilation**
        - Go creates compiled programs
            - there is a garbage collector (GC)
            - there is no virtual machine
    - **efficient execution**
    - **ease of programming**
    - the purpose of Go
- What Go is good for
    - what Google does / web services at scale
    - networking
        - http, tcp, udp
    - concurrency / parallelism
    - systems
    - automation, command-line tools
    - crypto
    - image processing
- Guiding principles of design
    - expressive, comprehensible, sophisticated
    - clean, clear, easy to read
- Companies using go
    - Google, YouTube, Google Confidential, Docker, Kubernetes, InfluxDB, Twitter, Apple
- The creator of Node.js has abandoned Node in favor of Go

files: Why Go - The Presentation

- **file can be located in the MISCELLANEOUS RESOURCES lecture on UDEMY**

video: 002

How to succeed

Understanding what has made others successful can help you become successful. These are principles which have helped me become successful. I learned these principles from others and from my own experience. I share these **principles to help you succeed in this course and in life:**

- Grit - Angela Duckworth
- Time on task
- Focus
    - Bill Gates & Warren Buffett
- Bill Gates, "If you want to be successful, get in front of what's coming and let it hit you."
- Habits of effective people
    - **file can be located in the MISCELLANEOUS RESOURCES lecture on UDEMY**
- my teachers
    - drop by drop, the bucket gets filled
    - persistently, patiently, you are bound to succeed

video: 003

Course resources

The course outline is part of the course. Please **read all of the descriptions of the videos in the course outline.** This is part of the learning process. When you read the descriptions:

- the concepts you are learning will be reinforced
- you **learn the material more quickly**

In addition, I sometimes provide additional information in the course descriptions. Sometimes I record a lecture, then remember that there is a resource or another piece of information which you should know. Some of these resources and extra pieces of information I include are very valuable

.

YOU CAN DOWNLOAD THE COURSE OUTLINES AND RESOURCES FROM WITHIN UDEMY

video: 004

Documentation

- **official documentation**

    - language spec
    - effective go
- **NOTE:**
    - **golang.org changed to go.dev/**
    - **godoc.org changed to pkg.go.dev/**
- **go.dev/ vs pkg.go.dev/**
    - go.dev/
        - language
        - standard library
    - pkg.go.dev/
        - standard library AND third-party packages

video: 005

Asking questions

This is THE BEST AND QUICKEST place to get your questions answered

- https://forum.golangbridge.org/

video: 005a

Accelerate learning

You can **increase the speed of videos when you watch them**. Not everyone knows this. This is something you should include in the beginning of all of your courses. Watching videos quickly helps many students. It's not for everybody, but it works for a lot of people. You need your students to know about this. You can also turn on the **"tools / document outline" for our course outline**.

video: 006

Your development environment

This section will show you how to set up your development environment. After this section, for the first part of this course, we will write code using an **online editor.** If you are new to programming, you can skip this section and come back to it later. If you are an experienced dev, you can set up your environment then write code using an integrated development environment (IDE) editor.

The terminal

- terminology
    - GUI = graphical user interface
    - CLI = command line interface - **command line**

        - **terminal** = text input/output environment; console = physical terminal
            - unix / linux / mac
                - **shell / bash / terminal**
            - windows
                - **command prompt / windows command / cmd / dos prompt**

video: 007a

Bash on Windows

- linux on Windows
    - developer features
        - Linux subsystem for Windows (beta)
    - bash
    - article with steps here and another article
- https://git-scm.com/

video: 007b

Shell / bash commands I

- shell / bash commands
    - **pwd**
    - **ls**
    - **ls -la**
        - permissions
            - owner, group, world
            - r, w, x
            - 4, 2, 1

d = directory
rwxrwxrwx = owner, group, world

	owner	group	bytes	last modification	hidden & name

```
total 72
drwxr-xr-x+ 20 spockmcleod  staff    680 Jul 31 15:44 .
drwxr-xr-x   5 root         admin    170 Dec 30  2016 ..
-r--------   1 spockmcleod  staff      7 Dec 30  2016 .CFUserTextEncoding
-rw-r--r--@  1 spockmcleod  staff  14340 Aug  1 06:52 .DS_Store
drwx------  19 spockmcleod  staff    646 Aug  1 07:07 .Trash
drwxr-xr-x  14 spockmcleod  staff    476 Jul 26 13:56 .atom
-rw-------   1 spockmcleod  staff  10321 Aug  1 07:16 .bash_history
drwx------  41 spockmcleod  staff   1394 Aug  1 07:16 .bash_sessions
drwx------   3 spockmcleod  staff    102 Aug 15  2016 .cups
-rw-------   1 spockmcleod  staff   1024 Dec 30  2016 .rnd
drwx------   4 spockmcleod  staff    136 Feb 15 14:48 Applications
drwx------+  4 spockmcleod  staff    136 Jul 26 13:16 Desktop
drwx------+  9 spockmcleod  staff    306 Jul 31 16:43 Documents
drwx------+  4 spockmcleod  staff    136 Jul 31 10:12 Downloads
```

- **cd**
- **cd ../**
- **mkdir**
- **touch**
  - touch temp.txt
- **nano temp.txt**
- **cat temp.txt**
- clear
  - **command + k**
  - **clear**
- **chmod**
  - chmod options permissions filename
  - chmod 777 temp.txt
- env
- **rm <file or folder name>**
  - **rm -rf <file or folder name>**
- .bash_profile & .bashrc
  - **.bash_profile** is executed for login shells, while **.bashrc** is executed for interactive non-login shells. When you login (type username and password) via console, either sitting at the machine, or remotely via ssh: .bash_profile is executed to configure your shell before the initial command prompt.
- grep
  - cat temp2.txt | grep enter
  - ls | grep -i documents
- all commands

video: 007c

Shell / bash commands II

- shell / bash commands
    - pwd
    - ls
    - ls -la
        - permissions
            - owner, group, world
            - r, w, x
            - 4, 2, 1

d = directory
rwxrwxrwx = owner, group, world

```
                    owner             group       bytes   last modification   hidden & name
total 72
drwxr-xr-x+ 20 spockmcleod  staff    680 Jul 31 15:44 .
drwxr-xr-x   5 root         admin    170 Dec 30  2016 ..
-r--------   1 spockmcleod  staff      7 Dec 30  2016 .CFUserTextEncoding
-rw-r--r--@  1 spockmcleod  staff  14340 Aug  1 06:52 .DS_Store
drwx------  19 spockmcleod  staff    646 Aug  1 07:07 .Trash
drwxr-xr-x  14 spockmcleod  staff    476 Jul 26 13:56 .atom
-rw-------   1 spockmcleod  staff  10321 Aug  1 07:16 .bash_history
drwx------  41 spockmcleod  staff   1394 Aug  1 07:16 .bash_sessions
drwx------   3 spockmcleod  staff    102 Aug 15  2016 .cups
-rw-------   1 spockmcleod  staff   1024 Dec 30  2016 .rnd
drwx------   4 spockmcleod  staff    136 Feb 15 14:48 Applications
drwx------+  4 spockmcleod  staff    136 Jul 26 13:16 Desktop
drwx------+  9 spockmcleod  staff    306 Jul 31 16:43 Documents
drwx------+  4 spockmcleod  staff    136 Jul 31 10:12 Downloads
```

    - cd
    - cd ../
    - mkdir
    - touch
        - touch temp.txt
    - nano temp.txt
    - cat temp.txt
    - clear
        - command + k
        - clear
    - chmod
        - chmod options permissions filename
        - chmod 777 temp.txt
    - env
    - rm <file or folder name>
        - rm -rf <file or folder name>

    - .bash_profile & .bashrc
        - **.bash_profile** is executed for login shells, while **.bashrc** is executed for interactive non-login shells. When you login (type username and password) via console, either sitting at the machine, or remotely via ssh: .bash_profile is executed to configure your shell before the initial command prompt.
    - grep
        - cat temp2.txt | grep enter
        - ls | grep -i documents
    - all commands

video: 007d

Installing Go

- download go
- go commands
    - **go version**
    - **go env**
    - **go help**
- checksum

video: 007e

Go modules intro

Starting in Go 1.13, module mode will be the default for all development. Here is what this means.

- version go 1.13

video: 007e2

Go workspace

- one folder - any name, any location
    - bin
    - pkg
    - src
        - github.com
            - <github.com username>
                - folder with code for project / repo
                - folder with code for project / repo
                - folder with code for project / repo
                - folder with code for project / repo
                - ...
                - folder with code for project / repo
- namespacing

- **go get**
    - package management
- GOPATH
    - points to your go workspace
- GOROOT
    - points to your binary installation of Go

video: 007f

Environment variables

- Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They are part of the environment in which a process runs.
- .bash_profile & .bashrc
    - **.bash_profile** is executed for login shells, while **.bashrc** is executed for interactive non-login shells. When you login (type username and password) via console, either sitting at the machine, or remotely via ssh: .bash_profile is executed to configure your shell before the initial command prompt.


```
Todds-MacBook-Pro:~ toddmcleod$ go env
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/toddmcleod/go"
GORACE=""
GOROOT="/usr/local/go"
GOTOOLDIR="/usr/local/go/pkg/tool/darwin_amd64"
GCCGO="gccgo"
CC="clang"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-arguments -fmessage-length=0 -gno-record-
gcc-switches -fno-common"
CXX="clang++"
CGO_ENABLED="1"
PKG_CONFIG="pkg-config"
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
Todds-MacBook-Pro:~ toddmcleod$ echo $PATH
/Users/toddmcleod/mongodb/bin:/Users/toddmcleod/google-cloud-sdk/bin:/usr/local/bin:/usr/bin:/bin:/usr/sb
in:/sbin:/usr/local/go/bin:/Users/toddmcleod/go/bin
Todds-MacBook-Pro:~ toddmcleod$
```

video: 007g

IDE's

- VS Code
- Goland
- Atom.io
- Sublime

Gogland

https://www.jetbrains.com/go/

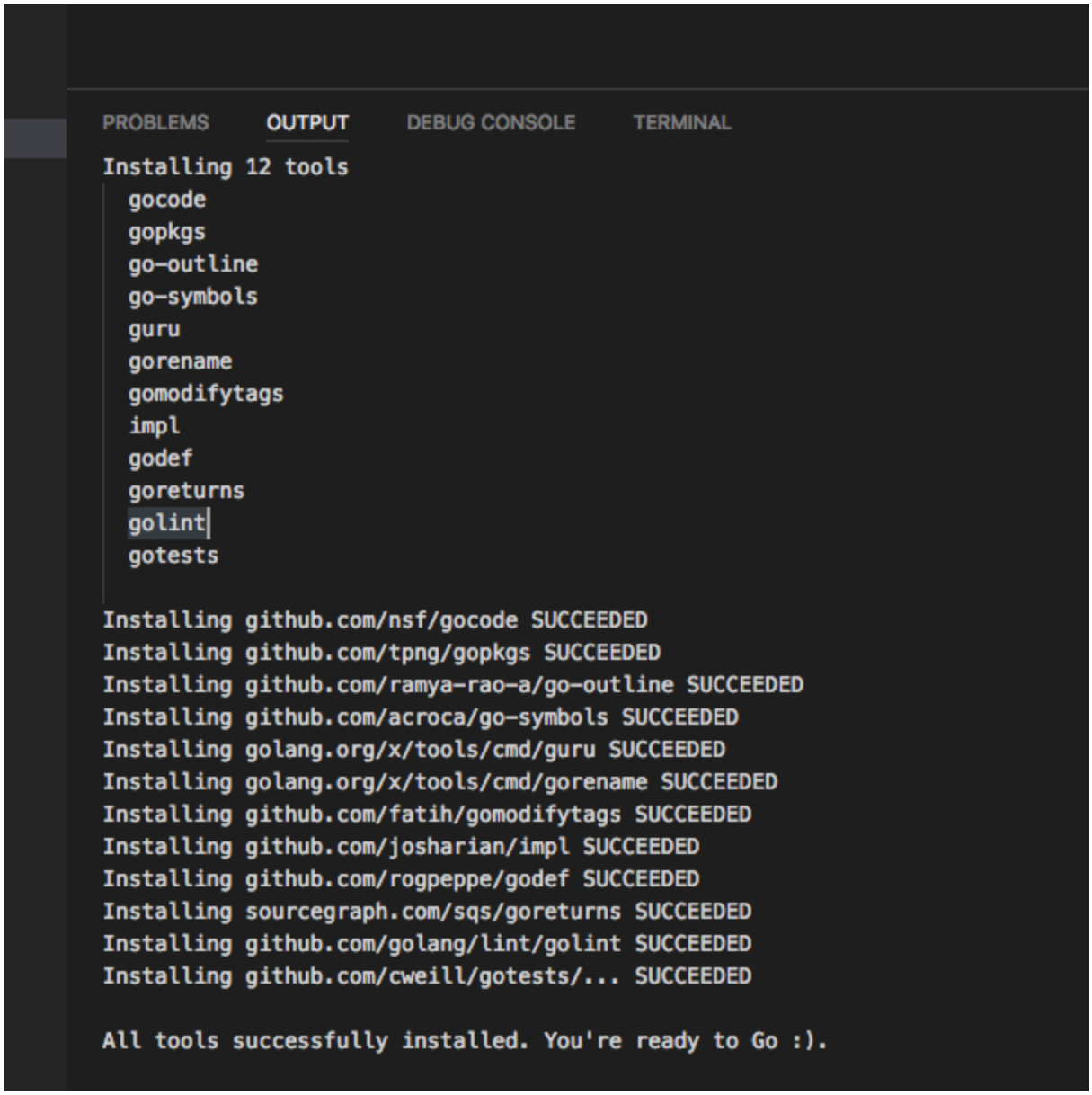
- bitstream vera sans mono
    - **file can be located in the MISCELLANEOUS RESOURCES lecture on UDEMY**
- McLeod's goland settings
    - **file can be located in the MISCELLANEOUS RESOURCES lecture on UDEMY**
- **go get**
    - go get -d github.com/GoesToEleven/go-programming
    - go get -d github.com/GoesToEleven/GolangTraining
    - go get -d github.com/GoesToEleven/golang-web-dev

VS CODE

https://code.visualstudio.com/

as always, things change fast. This has been deprecated. For VS CODE golang setup, use this instead

https://code.visualstudio.com/docs/languages/go



video: 007h

Go commands

- **go version**
- **go env**
- **go help**
- **go fmt**
  - **./...**
- **go run**
  - needs a file name, eg, go run main.go
  - go run <file name>

    - go run *.go
- **go build**
    - for an executable:
        - builds the file
        - reports errors, if any
        - if there are no errors, it puts an executable into the current folder
    - for a package:
        - builds the file
        - reports errors, if any
        - throws away binary
- **go install**
    - for an executable:
        - compiles the program (builds it)
        - names the executable
            - mac: the folder name holding the code
            - windows: file name
        - puts the executable in **workspace / bin**
            - $GOPATH / bin
    - for a package:
        - compiles the package (builds it)
        - puts the executable in **workspace / pkg**
            - $GOPATH / pkg
        - makes it an archive file
- **flags**
    - **-race**

video: 007i

Github repos

- creating a repo
    - create a repo on github.com
    - create a folder with the same name on your computer
        - put this in your goworkspace under
            - github.com
                - your user name
                    - your repo name
    - run “git init” on this folder
        - this create a git repository in the folder
    - add a file to this folder
        - “.gitignore” is a great file to add
    - have git track this file
        - git add --all
    - commit this change

        - git commit -m “here’s some commit message”
    - follow the instructions from github, from when you created your repo, to connect the repo ON YOUR COMPUTER with the repo on GITHUB - with your repo name instead of mine:
        - `git remote add origin git@github.com:GoesToEleven/tttttt.git`
          `git push -u origin master`
- git commands
    - git status
    - git add --all
    - git commit -m “some message”
    - git push

video: 007j

Github explored

- cloning a repo
    - git clone <repo>
- ssh
    - mac / bash / shell
        - ssh-keygen -t rsa
            - id_rsa
                - Your private key is saved to the id_rsa file in the .ssh directory and is used to verify the public key you use
            - id_rsa.pub
                - This is your public key. It is public. You can share it with the world.

video: 007k

Package management

When creating software today, you can use code that other people have written. You can call this other code ***packages, module, libraries, or dependencies.*** If you use other people’s code in ***your*** code, then ***your*** code is ***dependent*** upon other people’s code. Your code now has dependencies. ***Managing these packages / libraries / modules / dependencies (call it what you will) is what packagement management is all about.*** Here’s a great article by Russ Cox on it all:

- https://research.swtch.com/deps

Why you should be wary of other’s code:

- https://en.wikipedia.org/wiki/Npm_(software)#History LEFT-PAD

video: 007l package management

Go modules overview

- The Go Blog - Using Go Modules

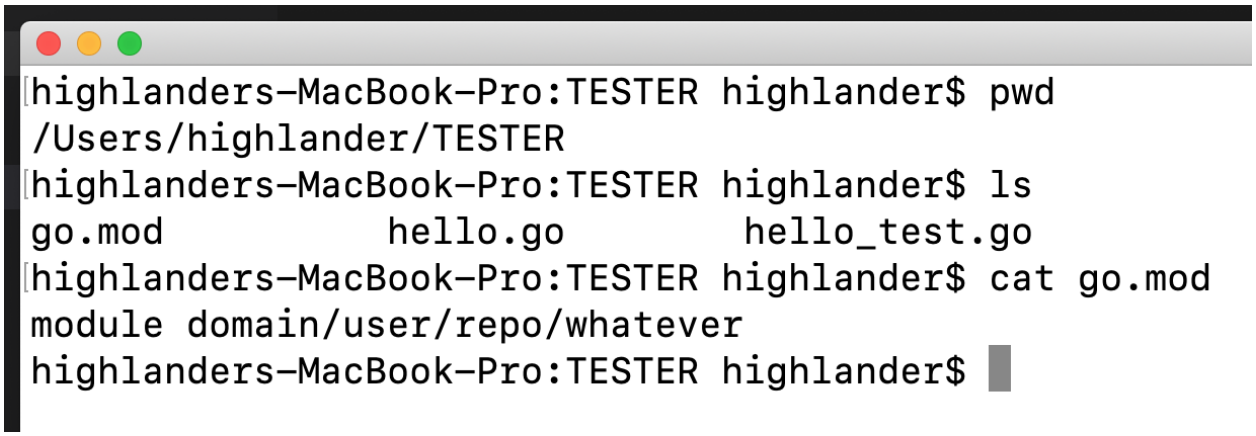
- https://blog.golang.org/using-go-modules

video: 007m go mods default

Creating a go module

You create a go module "workspace" using this command:

- go mod init example.com/hello
  - where "example.com/hello" is any text you want, ***and it is a good idea to use some name-spacing convention***



video: 007n creating module

Adding a dependency

You can add a dependency by importing a third-party package. To see all of the dependencies, use this command:

- go list -m all

video: 007o add dependency

Upgrading dependencies

You can upgrade dependencies, and also specify specific versions, of software that your code depends on.

- grabs latest version of imports
- to get a specific version, use "@" in your go get
  - for instance: go get github.com/gorilla/mux@v1.7.3
- "@" will support
  - release version
  - branch
  - commit #s

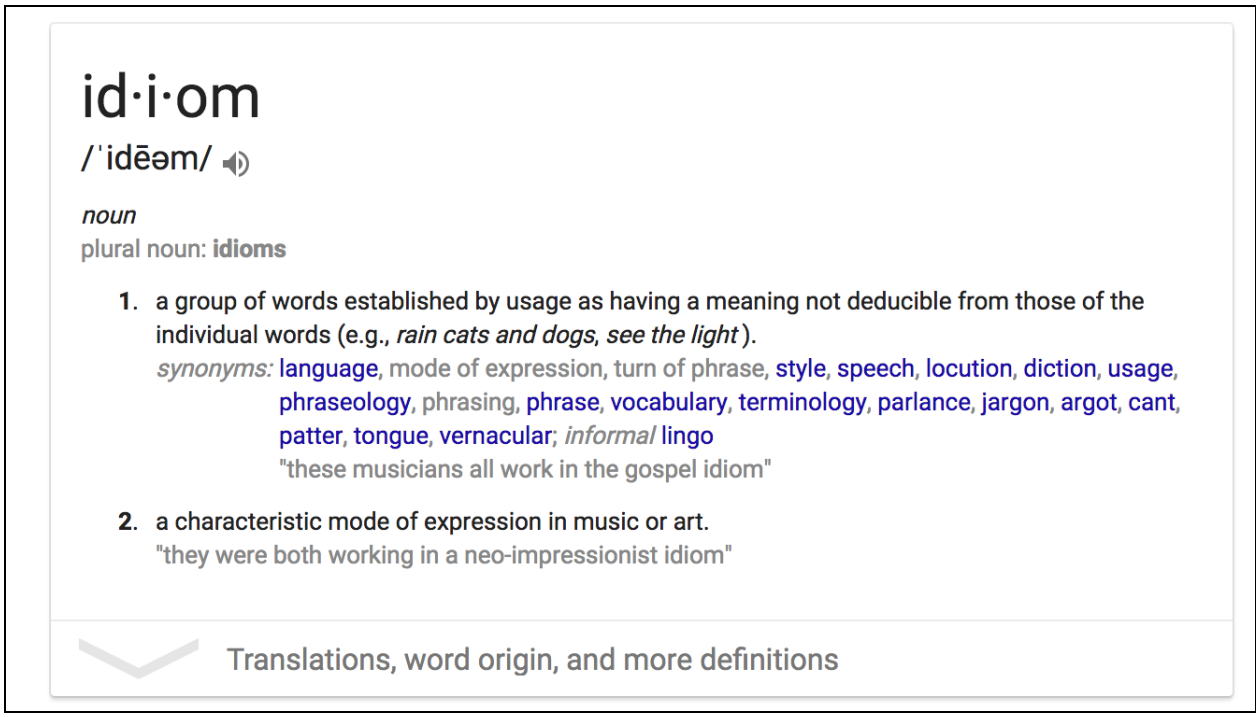
go list -m -versions <import>

video: 007p upgrad dep

Variables, values, & type

Playground

- share
- import
- format
- run
    - https://forum.golangbridge.org/
- **"idiomatic go"**
    - idioms are patterns of speech
    - when someone writes "idiomatic Go" they are writing Go code in the way Go code community writes code



id·i·om

/ˈidēəm/

noun

plural noun: **idioms**

1. a group of words established by usage as having a meaning not deducible from those of the individual words (e.g., *rain cats and dogs*, *see the light*).
   synonyms: language, mode of expression, turn of phrase, style, speech, locution, diction, usage, phraseology, phrasing, phrase, vocabulary, terminology, parlance, jargon, argot, cant, patter, tongue, vernacular; *informal* lingo
   "these musicians all work in the gospel idiom"

2. a characteristic mode of expression in music or art.
   "they were both working in a neo-impressionist idiom"

Translations, word origin, and more definitions

video: 008

Hello world

- basic program structure:
    - package main
    - **func main**
        - **entry point to your program**
        - when your code exits func main, your program is over

code: https://play.golang.org/p/v3rrZLwEUC

video: 009

Introduction to packages

- variadic parameters
    - the "...<some type>" is how we signify a **variadic parameter**
    - the type "interface{}" is the empty interface
        - every value is also of type "**interface{}**"
    - so the parameter "...interface{}" means "give me as many arguments of any type as you'd like
- throwing away returns
    - use **the "_" underscore character** to throw away returns
- you can't have unused variables in your code
    - this is code pollution
    - the compiler doesn't allow it
- we use this notation in Go
    - **package.Identifier**
        - ex: fmt.Println()
            - we would read that: "from package fmt I am using the Println func"
    - an identifier is the name of the variable, constant, func
- packages
    - code that is already written which you can use
    - imports

code: https://play.golang.org/p/30rOKO3qT8s
video: 009a

Short declaration operator

- terminology
    - **keywords**
        - these are words that a reserved for use by the Go programming language
            - they are sometimes called "reserved words"
            - you can't use a keyword for anything other than its purpose
    - **operator**
        - in "2 + 2" the "+" is the OPERATOR
        - an operator is a character that represents an action, as for example "+" is an arithmetic OPERATOR that represents addition
    - **operand**
        - in "2 + 2" the "2"s are OPERANDS
    - **statement**
        - In programming a statement is the smallest standalone element of a program that expresses some action to be carried out. It is an instruction

that commands the computer to perform a specified action. A program is formed by a sequence of one or more statements.
    - **expression**
        - in programming an expression is a combination of one or more explicit values, constants, variables, operators, and functions that the programming language interprets and computes to produce another value. For example, 2+3 is an expression which evaluates to 5.
- golang mascot
    - https://github.com/golang/go/tree/master/doc/gopher
        - Renne French
    - https://github.com/ashleymcnamara/gophers
    - https://github.com/egonelbre/gophers

code:

- **https://play.golang.org/p/1tW0Z2onP-**
- **https://play.golang.org/p/Y-jFwR11N3u**

video: 010

The var keyword

- **parens**

  ( )
- **curly braces**

  { }
- where var can be used
    - any place within the package
- scope
    - where a variable exists and is accessible
    - best practice: keep scope as “narrow” as possible

code:

- **https://play.golang.org/p/iXJaHEZztc**
- **https://play.golang.org/p/D0Sxev__J_i**

video: 011

Exploring type

- DECLARE a VARIABLE is of a certain TYPE it can only hold VALUES of that TYPE
- “Go suffers no fools.”
    - like “dead men tell no tales”
- var z int = 21
    - package scope
- **primitive data types**

    - In computer science, primitive data type is either of the following:
        - **a basic type** is a data type provided by a programming language as a basic building block. Most languages allow more complicated *composite types* to be constructed starting from basic types.
        - **a built-in type** is a data type for which the programming language provides built-in support.
    - In most programming languages, all basic data types are built-in. In addition, many languages also provide a set of composite data types. Opinions vary as to whether a built-in type that is not basic should be considered "primitive".
    - https://en.wikipedia.org/wiki/Primitive_data_type
- **composite data types**
    - In computer science, a **composite data type** or **compound data type** is any data type which can be constructed in a program using the programming language's primitive data types and other composite types. It is sometimes called a **structure** or **aggregate data type**, although the latter term may also refer to arrays, lists, etc. ***The act of constructing a composite type is known as composition***

code:

- **https://play.golang.org/p/gHld71M4uY**
- **https://play.golang.org/p/Lrl7oRoVzz**
- **https://play.golang.org/p/vuTD-E59Rb5**

video: 012

Zero value

- understanding **zero value**
    - false for booleans
    - 0 for integers
    - 0.0 for floats
    - "" for strings
    - nil for
        - pointers
        - functions
        - interfaces
        - slices
        - channels
        - maps
- use short declaration operator as much as possible
- use var for
    - zero value
    - package scope

code:

- **https://play.golang.org/p/v3ez_kTwjn**

- **https://play.golang.org/p/0nVApmU1hsK**

video: 013

The fmt package

- basic code setup
    - using var
        - using zero value
    - using short declaration operator
    - https://play.golang.org/p/oLxC8gHE0D
- %v
    - https://play.golang.org/p/51WOG55riI
- escaped characters like \n or \t
    - https://golang.org/ref/spec#Rune_literals
- **format printing**
    - https://play.golang.org/p/rqGVNVP5kI
    - we look at the difference between these funcs in the “fmt” package
        - group #1 - general printing to standard out
            - func Print(a ...interface{}) (n int, err error)
            - func Printf(format string, a ...interface{}) (n int, err error)
            - func Println(a ...interface{}) (n int, err error)t
        - group #2 = printing to a string which you can assign to a variable
            - func Sprint(a ...interface{}) string
            - func Sprintf(format string, a ...interface{}) string
            - func Sprintln(a ...interface{}) string
        - group #3 - printing to a file or a web server’s response
            - func Fprint(w io.Writer, a ...interface{}) (n int, err error)
            - func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
            - func Fprintln(w io.Writer, a ...interface{}) (n int, err error)

code:

- **https://play.golang.org/p/o36sAgq5BkT**

video: 014

Creating your own type

- **we can create our own type in Go**
    - https://play.golang.org/p/5lTs6Z6PWI

video: 015

Conversion, not casting

Go has a language to talk about its language. Terms have been jettisoned because they come with baggage. Re-envision programming. New words to talk about some concepts. We

don't talk about objects we talk about creating TYPES and VALUES of a certain TYPE. We don't talk about casting, we talk about CONVERSION and ASSERTION.

- https://play.golang.org/p/1NqUFF242Y

video: 016

Exercises - Ninja Level 1

Hands-on exercise #1

1. Using the short declaration operator, **ASSIGN** these **VALUES** to **VARIABLES** with the IDENTIFIERS "x" and "y" and "z"
    a. 42
    b. "James Bond"
    c. true
2. Now print the values stored in those variables using
    a. a single print statement
    b. multiple print statements

code: here's the solution: https://play.golang.org/p/yYXnWXIQNa
video: 017

Hands-on exercise #2

1. Use var to **DECLARE** three **VARIABLES**. The variables should have package level scope. Do not assign **VALUES** to the variables. Use the following **IDENTIFIERS** for the variables and make sure the variables are of the following TYPE (meaning they can store VALUES of that **TYPE**).
    a. identifier "x" type int
    b. identifier "y" type string
    c. identifier "z" type bool
2. in func main
    a. print out the values for each identifier
    b. The compiler assigned values to the variables. What are these values called?

code: here's the solution: https://play.golang.org/p/jzHwSlles9
video: 018

Hands-on exercise #3

Using the code from the previous exercise,

1. At the package level scope, assign the following values to the three variables
    a. for x assign 42
    b. for y assign "James Bond"
    c. for z assign true
2. in func main

a. use fmt.Sprintf to print all of the VALUES to one single string. ASSIGN the returned value of TYPE string using the short declaration operator to a VARIABLE with the IDENTIFIER “s”
b. print out the value stored by variable “s”

code: here’s the solution: https://play.golang.org/p/QFctSQB_h3
video: 019

Hands-on exercise #4

- FYI - nice documentation and new terminology “**underlying type**”
  - https://golang.org/ref/spec#Types

For this exercise

1. Create your own type. Have the underlying type be an int.
2. create a VARIABLE of your new TYPE with the IDENTIFIER “x” using the “VAR” keyword
3. in func main
   a. print out the value of the variable “x”
   b. print out the type of the variable “x”
   c. assign 42 to the VARIABLE “x” using the “=” OPERATOR
   d. print out the value of the variable “x”

code: here’s the solution: https://play.golang.org/p/snm4WuuYmG
video: 020

Hands-on exercise #5

Building on the code from the previous example

1. at the package level scope, using the “var” keyword, create a VARIABLE with the IDENTIFIER “y”. The variable should be of the UNDERLYING TYPE of your custom TYPE “x”
   a. eg:

```
type hotdog int

var x hotdog
var y int
```

2. in func main
   a. this should already be done
      i. print out the value of the variable “x”
      ii. print out the type of the variable “x”
      iii. assign your own VALUE to the VARIABLE “x” using the “=” OPERATOR

iv. print out the value of the variable “x”

b. now do this
    i. now use CONVERSION to convert the TYPE of the VALUE stored in “x” to the UNDERLYING TYPE
        1. then use the “=” operator to ASSIGN that value to “y”
        2. print out the value stored in “y”
        3. print out the type of “y”

code: here’s the solution: https://play.golang.org/p/cj8RrYgBOD
video: 021

Quiz for "variables, values & type"

Hands-on exercise #6

An explanation of the quiz questions and their answers is provided.
video: 022

Programming fundamentals

Bool type

1. A **bool** is a binary TYPE having two possible values of either “true” and “false.”
2. When you use an **equality comparison operator**, this is an expression which will evaluate to a boolean value
    a. **==**
    b. **<=**
    c. **>=**
    d. **!=**
    e. **<**
    f. **>**

code

1. **https://play.golang.org/p/WmPsOkVzwS**
2. **https://play.golang.org/p/inS_7F0HdC**

video: 023

How computers work

- computers run on electricity
- electricity has two discrete states: ON and OFF
- we can create **coding schemes** for “on” or “off” states

files: presentation
video: 024

Numeric types

- integers
    - numbers without decimals
        - aka, whole number
        -
    - int & uint
        - “implementation-specific sizes”
    - all numeric types are distinct except
        - **byte** which is an alias for **uint8**
        - **rune** which is an alias for **int32**
    - types are unique
        - “this is static programming language
        - “Conversions are required when different numeric types are mixed in an expression or assignment. For instance, int32 and int are not the same type even though they may have the same size on a particular architecture” (source)
    - rule of thumb: just use **int**
- floating point
    - numbers with decimals
        - aka, real numbers
    - rule of thumb: just use **float64**
- nice reading - Caleb Doxsey’s book

code:

- https://play.golang.org/p/OdWUH8uva6
- https://play.golang.org/p/0JpmCYezs1
- this does not run: https://play.golang.org/p/O7nFEn8nXz
- int8
    - https://play.golang.org/p/IcOtgm6YKA
    - does not work: https://play.golang.org/p/YbwTa1YT4i
    - https://play.golang.org/p/exwG0ijjRf
    - does not work: https://play.golang.org/p/sy16rgifWF

runtime package

- GOOS
- GOARCH
- https://play.golang.org/p/1vp5DImIMM

video: 025

String type

- golang blog post on strings

- “A string value is a (possibly empty) sequence of bytes. Strings are immutable: once created, it is impossible to change the contents of a string.”
- Rob Pike’s blog post on string

code:

- nice examples - lecture prep:
  - https://play.golang.org/p/LUbFEJEope
  - https://play.golang.org/p/JjWLMcAsCU
- live coding:
  - https://play.golang.org/p/W6iH-Re6Zp

video: 026

Numeral systems

As humans, we have many different systems for expressing the quantities of something. Using the decimal numeral system, we might say we have 7 oranges; or 14 carrots; or 42 dollars. Other numeral systems which are used in programming include the **binary numeral system** and the **hexadecimal numeral system.**

video: 027

Constants

- a simple, unchanging value
- Only exist at compile time.
- there are **TYPED** and **UNTYPED** constants
  - `const hello = "Hello, World"`
  - `const typedHello string = "Hello, World"`
- UNTYPED constant
  - a constant value that does not yet have a fixed type
    - “constant of a **kind**”
    - not yet forced to obey the strict rules that prevent combining differently typed values
  - An untyped constant can be implicitly converted by the compiler.
- It is this notion of an *untyped* constant that makes it possible for us to use constants in Go with great freedom.
  - This is useful, for instance
    - what is the type of 42?
      - int?
      - uint?
      - float64?
    - if we didn’t have UNTYPED constants (constants of a kind), then we would have to do conversion on every literal value we used

        - and that would suck
- code used in video
    - https://play.golang.org/p/-ZwrDDimgH
- lecture prep
    - https://play.golang.org/p/lVURPQe-N4

video: 028

Iota

Within a constant declaration, the predeclared identifier iota represents **successive untyped integer constants**. It is reset to 0 whenever the reserved word const appears in the source. It can be used to construct a set of related constants:

- code used in video
    - https://play.golang.org/p/_cSkz_b28t
- lecture prep
    - https://play.golang.org/p/YOabnTj5OI
    - https://play.golang.org/p/c5SmcFzzBM

video: 029

Bit shifting

Bit shifting is when you **shift binary digits to the left or right.** We can use bit shifting, along with iota, to build some creative constants.

nice article

- https://medium.com/learning-the-go-programming-language/bit-hacking-with-go-e0acee258827

code in video

- https://play.golang.org/p/YnMevatXlP
- https://play.golang.org/p/q_IGHQ2am4
- https://play.golang.org/p/IwNVIOcrLG

prep code:

- https://play.golang.org/p/3oxB39hYJ_
- https://play.golang.org/p/7MOnbhx4R4
- this is instructive, shows all bits shifting: https://play.golang.org/p/DK6Ub7Sotx

video: 030

Exercises - Ninja Level 2

Hands-on exercise #1

Write a program that prints a number in **decimal, binary, and hex**
solution: https://play.golang.org/p/bAQxcEuK8O

video: 031

Hands-on exercise #2

Using the following operators, write expressions and assign their values to variables:

g. **==**
h. **<=**
i. **>=**
j. **!=**
k. **<**
l. **>**

Now print each of the variables.
solution: https://play.golang.org/p/76R-poSzaY
video: 032

Hands-on exercise #3

Create **TYPED and UNTYPED constants**. Print the values of the constants.
solution: https://play.golang.org/p/NutvJXWUx2
video: 033

Hands-on exercise #4

Write a program that

- assigns an int to a variable
- prints that int in decimal, binary, and hex
- **shifts the bits** of that int over 1 position to the left, and assigns that to a variable
- prints that variable in decimal, binary, and hex

solution: https://play.golang.org/p/Ms964T8SbH
video: 034

Hands-on exercise #5

Create a variable of type string using a **raw string literal**. Print it.
solution: https://play.golang.org/p/dLy36A-V-w
video: 035

Hands-on exercise #6

Using **iota**, create 4 constants for the **NEXT** 4 years. Print the constant values.
solution: https://play.golang.org/p/MDLF3v5EGT
video: 036

Quiz for Fundamentals of Programming

Hands-on exercise #7

In this hands-on exercise, I will go through the quiz with you!
video: 037

Control Flow

Understanding control flow

Computers read programs in a certain way, just like we read books in a certain way. When we, as humans, read books, in Western culture, we read from front to back, left to right, top to bottom. When computers read code in a program, they read it from top to bottom. This is known as reading it in SEQUENCE. This is also known as SEQUENTIAL control flow. In addition to sequential control flow, there are two other statements which can affect how a computer reads code. A computer might hit a LOOP control flow. If it hits one of these, it will loop over the code in a specified manner. Loop control flow is also known as ITERATIVE control flow. Finally there is also CONDITIONAL control flow. When the computer hits a CONDITION, like an “if statement” or a “switch statement” the computer will take some course of action depending upon some condition. So the three ways a computer reads code are: **SEQUENTIAL, LOOP, CONDITIONAL**

- sequence
- loop / iterative
    - for loop
        - init, cond, post
    - bool (while-ish)
        - infinite
    - do-while-ish
        - break
    - continue
    - nested
- conditionals
    - switch / case / default statements
        - no default fall-through
        - creating fall-through
        - multiple cases
        - cases can be expressions
            - evaluate to true, they run
        - type

        - if
            - the not operator
                - !
            - initialization statement
            - if / else
            - if / else if / else
            - if / else if / else if / … / else

video: 038

Loop - init, condition, post

- for loop
    - **initialization, condition, post**

code:

- init,condition, post https://play.golang.org/p/Wp5cT2IaMx
- for: https://play.golang.org/p/A0GUGbqi9_

video: 039

Loop - nested loops

- for loop
    - **nested loops**
- code:
    - https://play.golang.org/p/0N-eThFPVZ
    - https://play.golang.org/p/RVSKbHUQLb

video: 040

Loop - for statement

There are three ways you can do loops in Go - they all just use the **“for” keyword**:

- for init; condition; post { }
- for condition { }
- for { }

Reading documentation for the “for” statement

- language spec
- effective go

code

- https://play.golang.org/p/IsqvsdiIBH
- https://play.golang.org/p/egX34wR3wX

video: 041

Loop - break & continue

for loop

- break
- continue

finding a remainder, also known as a **modulus**

- **%**

code:

- remainder: https://play.golang.org/p/_BNQa7c8d8
- break & continue: https://play.golang.org/p/uqh2SDENAE

video: 042

Loop - printing ascii

We can **use format printing** to print out the

- decimal value with **%d**
- hex value with **%#x**
- unicode code point with **%#U**
- a tab with **\t**
- a new line with **\n**

code:

- https://play.golang.org/p/SnzlisWesT

student code:

- **https://play.golang.org/p/BzCszCBEn7W**

video: 043

Conditional - if statement

If Statements

- bool
  - true
  - false
- the not operator
  - **!**
- **initialization statement**

code:

- **https://play.golang.org/p/vz-qKTA7a2**
- **initialization statement: https://play.golang.org/p/CxpQu6fLzN**

video: 044

Conditional - if, else if, else

- if / else
  - https://play.golang.org/p/RRxtR_QTd3
- if / else if / else
  - https://play.golang.org/p/4_E3QSKox0
- **if / else if / else if / … / else**

- https://play.golang.org/p/YsNIxfikEq

video: 045

Loop, conditional, modulus

We iterate in life to improve. This is true for whatever we're doing. We write the code with errors before we write the code without errors. I had a teacher who called this the "perfection of imperfection." My teacher would say to me, "You are perfect just the way you are, and there is room for improvement." The point being - the code we wrote in an earlier video to print out even numbers can be done better. In this video, we are going to do it better. We are going to write code that is more clear. This will serve as a review of the concepts we have been learning: **loops, conditional if statements, and the modulo operator.**

video: 046

Conditional - switch statement

Switch Statements

- switch / case / default statements
  - no default fall-through
  - creating fall-through
  - multiple cases
  - cases can be expressions
    - evaluate to true, they run

code:

- switch on boolean value: https://play.golang.org/p/BFRFxQKT9H
- no default fallthrough: https://play.golang.org/p/1EtC2k2rvX
- funky fallthrough: https://play.golang.org/p/0lOxS35E_e
- default https://play.golang.org/p/-Ybc4TVwO4
- switch on a value - which one does it match?: https://play.golang.org/p/hw35DavFFt
- switch on multiple matches for a case: https://play.golang.org/p/De_2ZfSaqN

video: 047

Conditional - switch statement documentation

Being comfortable with reading documentation is important. By spending a little time together looking through Go's documentation, you will see how to read and interpret the documentation and become more comfortable with it.

video: 048

Conditional logic operators

What do these expressions evaluate to:

- fmt.Println(**true && true**)
- fmt.Println(**true && false**)

- fmt.Println(**true || true**)
- fmt.Println(**true || false**)
- fmt.Println(**!true**)

code

- https://play.golang.org/p/ukFrlC66uv
- https://play.golang.org/p/WQ8PjCCC42

(source)

video: 049

Exercises - Ninja Level 3

Hands-on exercise #1

Print every number from 1 to 10,000

solution: https://play.golang.org/p/voDiuiDGGw

video: 050

Hands-on exercise #2

Print every rune code point of the uppercase alphabet three times. Your output should look like this:

```
65
        U+0041 'A'
        U+0041 'A'
        U+0041 'A'
66
        U+0042 'B'
        U+0042 'B'
        U+0042 'B'
 ... through the rest of the alphabet characters
```

solution: https://play.golang.org/p/1OjnCX1D5H

video: 051

Hands-on exercise #3

Create a for loop using this syntax

- for condition { }

Have it print out the years you have been alive.

solution: https://play.golang.org/p/tnyqBPJ-i5

video: 052

Hands-on exercise #4

Create a for loop using this syntax

- for { }

Have it print out the years you have been alive.
solution: https://play.golang.org/p/9VpnB-I1Pz
video: 053

Hands-on exercise #5

Print out the remainder (modulus) which is found for each number between 10 and 100 when it is divided by 4.
solution: https://play.golang.org/p/ohfJOW9euy
video: 054

Hands-on exercise #6

Create a program that shows the “if statement” in action.
solution: https://play.golang.org/p/DpZ_FLfn5s
video: 055

Hands-on exercise #7

Building on the previous hands-on exercise, create a program that uses “else if” and “else”.
solution: https://play.golang.org/p/IDnrJpE7vT
video: 056

Hands-on exercise #8

Create a program that uses a switch statement with no switch expression specified.
solution: https://play.golang.org/p/YpPgkWGqKY
video: 057

Hands-on exercise #9

Create a program that uses a switch statement with the switch expression specified as a variable of TYPE string with the IDENTIFIER “favSport”.
solution: https://play.golang.org/p/h-8FnjpzWB
video: 058

Hands-on exercise #10

Write down what these print:

- fmt.Println(**true && true**)

- fmt.Println(**true && false**)
- fmt.Println(**true || true**)
- fmt.Println(**true || false**)
- fmt.Println(**!true**)

solution:
video: 059

Grouping data

- **array**
  - a numbered sequence of elements of a single type
  - does not change in size
  - **used for Go internals; generally not recommended for your code**
  - https://golang.org/ref/spec#Array_types
- **slice**
  - built on top of an array
  - **holds values of the same type**
  - changes in size
  - has a length and a capacity
  - https://golang.org/ref/spec#Slice_types
- **map**
  - **key / value storage**
  - an **unordered** group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type.
  - https://golang.org/ref/spec#Map_types
- **struct**
  - a data structure
  - a composite type
  - allows us to **collect values of different types together**
  - https://golang.org/ref/spec#Struct_types

Array

Arrays are mostly used as a building block in the Go programming language. In some instances, we might use an array, but mostly the recommendation is to **use slices instead.**

code

- https://play.golang.org/p/f-7aufl2DO

video: 060

Slice - composite literal

A SLICE holds VALUES of the same TYPE. If I wanted to store all of my favorite numbers, I would use a slice of int. If I wanted to store all of my favorite foods, I would use a slice of string. We will **use a COMPOSITE LITERAL to create a slice.** A composite literal is created by having the TYPE followed by CURLY BRACES and then putting the appropriate values in the curly brace area.

code

- https://play.golang.org/p/XtUEPJFgqD

video: 061

Slice - for range

We can **loop over the values in a slice with the range clause.** We can also access items in a slice by index position.

code

- https://play.golang.org/p/O7cCALNFsH

video: 062

Slice - slicing a slice

We can slice a slice which means that we can **cut parts of the slice away.** We do this with the colon operator.

code

- https://play.golang.org/p/AXGTEEn92M

video: 063

Slice - append to a slice

To append values to a slice, we use the special built in function append. This function returns a slice of the same type.

code

- https://play.golang.org/p/oQnjP5Ka3F

video: 064

Slice - deleting from a slice

We can **delete from a slice using both append and slicing.** This is a wonderful and elegant example of why Go is great and how Go provides ease of programming.

code:

https://play.golang.org/p/VTZ2Bof6bN

video: 065

Slice - make

Slices are built on top of arrays. A slice is dynamic in that it will grow in size. The underlying array, however, does not grow in size. When we create a slice, **we can use the built in function make to specify how large our slice should be and also how large the underlying array should be.** This can enhance performance a little bit.

- make([]T, length, capacity)
- make([]int, 50, 100)

code

- https://play.golang.org/p/07hH1b-hvD

video: 066

Slice - multi-dimensional slice

A multi-dimensional slice is like a spreadsheet. You can have **a slice of a slice of some type.** Does that sound confusing? Watch this video and it will all be clarified.

code

- https://play.golang.org/p/S4cyB89Zpm

video: 067

Slice - underlying array

Underlying every slice is an array. A slice is actually a data structure which has three parts:

1. a pointer to an array
2. len
3. cap

In this video, we will explore the relationship between the slice and the underlying array.

code

- a new underlying array is allocated
  - https://play.golang.org/p/XE9l_on3Va
- the same array is used for TWO slices
  - https://play.golang.org/p/Ah2t7mgTn4
  - you can also “throw away” the variable and the same thing happens
    - https://play.golang.org/p/NL8p7Z8R3E
- looking at the LEN & CAP of a slice
  - https://play.golang.org/p/GvEnKl3x-A
-

video: 067-b

Map - introduction

A map is a key-value store. This means that you store some value and you access that value by a key. For instance, I might store the value “phoneNumber” and access it be the key

“friendName”. This way I could enter my friend’s name and have the map return their phone number. The syntax for a map is map[key]value. The key can be of any type which allows comparison (eg, I could use a string or an int, for example, as I can compare if two strings are equal, or if two ints are equal). It is important to note that maps are unordered. If you print out all of the keys and values in a map, they will print out in random order. The **comma ok idiom** is also covered in this video. **A map is the perfect data structure when you need to look up data fast.**

code:

- **https://play.golang.org/p/TJajJyrUo5**

video: 068

Map - add element & range

Here is how we **add an element to a map**. I also show you **how to use the range loop to print out a map’s** keys and values.

code:

- **https://play.golang.org/p/RTuBRiW087**

video: 069

Map - delete

You delete an entry from a map using **delete(<map name>, “key”)**. No error is thrown if you use a key which does not exist. To confirm you delete a key/value, verify that the key/value exists with the comma ok idiom.

code:

- **https://play.golang.org/p/t5g_-8wgOL**

video: 070

Exercises - Ninja Level 4

Hands-on exercise #1

- Using a COMPOSITE LITERAL:
  - create an ARRAY which holds 5 VALUES of TYPE int
  - assign VALUES to each index position.
- Range over the array and print the values out.
- Using format printing
  - print out the TYPE of the array

solution: https://play.golang.org/p/tD0SzV3hdf

video: 071

Hands-on exercise #2

- Using a COMPOSITE LITERAL:
    - create a SLICE of TYPE int
    - assign 10 VALUES
- Range over the slice and print the values out.
- Using format printing
    - print out the TYPE of the slice

solution: https://play.golang.org/p/sAQeFB7Dls
video: 072

Hands-on exercise #3

Using the code from the previous example, use SLICING to create the following new slices which are then printed:

- [42 43 44 45 46]
- [47 48 49 50 51]
- [44 45 46 47 48]
- [43 44 45 46 47]

solution: https://play.golang.org/p/SGfiULXzAB
video: 073

Hands-on exercise #4

Follow these steps:

- start with this slice
    - x := []int{42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
- append to that slice this value
    - 52
- print out the slice
- in ONE STATEMENT append to that slice these values
    - 53
    - 54
    - 55
- print out the slice
- append to the slice this slice
    - y := []int{56, 57, 58, 59, 60}
- print out the slice

solution: https://play.golang.org/p/QUYhtSBaDS
video: 074

Hands-on exercise #5

To DELETE from a slice, we use APPEND along with SLICING. For this hands-on exercise, follow these steps:

- start with this slice
    - x := []int{42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
- use APPEND & SLICING to get these values here which you should ASSIGN to a variable “y” and then print:
    - [42, 43, 44, 48, 49, 50, 51]

solution: https://play.golang.org/p/u8zpHLfgSE
video: 075

Hands-on exercise #6

Create a slice to store the names of all of the states in the United States of America. Use **make** and **append** to do this. Goal: do not have the array that underlies the slice created more than once. What is the length of your slice? What is the capacity? Print out all of the values, along with their index position, without using the range clause. Here is a list of the 50 states:

` Alabama`, ` Alaska`, ` Arizona`, ` Arkansas`, ` California`, ` Colorado`, ` Connecticut`, ` Delaware`, ` Florida`, ` Georgia`, ` Hawaii`, ` Idaho`, ` Illinois`, ` Indiana`, ` Iowa`, ` Kansas`, ` Kentucky`, ` Louisiana`, ` Maine`, ` Maryland`, ` Massachusetts`, ` Michigan`, ` Minnesota`, ` Mississippi`, ` Missouri`, ` Montana`, ` Nebraska`, ` Nevada`, ` New Hampshire`, ` New Jersey`, ` New Mexico`, ` New York`, ` North Carolina`, ` North Dakota`, ` Ohio`, ` Oklahoma`, ` Oregon`, ` Pennsylvania`, ` Rhode Island`, ` South Carolina`, ` South Dakota`, ` Tennessee`, ` Texas`, ` Utah`, ` Vermont`, ` Virginia`, ` Washington`, ` West Virginia`, ` Wisconsin`, ` Wyoming`,

solution:

- **my first take on this project, which was an INCORRECT solution**
    - **https://go.dev/play/p/vE9BCWMA2ET**
- **instructive proof from a student which showed me the error of my ways :)**
    - **https://go.dev/play/p/szSlBwJG2S-**
- **HERE IS THE CORRECT SOLUTION**
    - **https://go.dev/play/p/ZcyFCyo7_XH**

It is good to think about the LEN and CAP. It is also good to think about the underlying array. You want to think about this so that you're thinking of memory use. Also, it is good to think about what got stored in what position. And it is good to remember that slices are dynamic - they can grow in size.
FROM THE SPEC:
The length of a slice s can be discovered by the built-in function len; unlike with arrays it may change during execution. The elements can be addressed by integer indices 0

through len(s)-1. The slice index of a given element may be less than the index of the same element in the underlying array.

A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

The array underlying a slice may extend past the end of the slice. The capacity is a measure of that extent: it is the sum of the length of the slice and the length of the array beyond the slice; a slice of length up to that capacity can be created by slicing a new one from the original slice. The capacity of a slice a can be discovered using the built-in function cap(a).
https://go.dev/ref/spec#Slice_types
video: 076

Hands-on exercise #7

Create a slice of a slice of string ([][]string). Store the following data in the multi-dimensional slice:

> "James", "Bond", "Shaken, not stirred"
> "Miss", "Moneypenny", "Hellooooooo, James."

Range over the records, then range over the data in each record.
solution: https://play.golang.org/p/FMM4c2PhZg
video: 077

Hands-on exercise #8

Create a map with a key of TYPE string which is a person's "last_first" name, and a value of TYPE []string which stores their favorite things. Store three records in your map. Print out all of the values, along with their index position in the slice.

> `bond_james`, `Shaken, not stirred`, `Martinis`, `Women`
> `moneypenny_miss`, `James Bond`, `Literature`, `Computer Science`
> `no_dr`, `Being evil`, `Ice cream`, `Sunsets`

solution: https://play.golang.org/p/nTzSlRa9_A
video: 078

Hands-on exercise #9

Using the code from the previous example, add a record to your map. Now print the map out using the "range" loop

solution: https://play.golang.org/p/_CkxAhRrDJ
video: 079

Hands-on exercise #10

Using the code from the previous example, delete a record from your map. Now print the map out using the “range” loop
solution: https://play.golang.org/p/TYI5EbjoeC
video: 080

Structs

Struct

A struct is an composite data type. (composite data types, aka, aggregate data types, aka, complex data types). **Structs allow us to compose together values of different types.**
code: https://play.golang.org/p/hNI_rSK-C6
video:081

Embedded structs

We can **take one struct and embed it in another struct.** When you do this the inner type gets promoted to the outer type.
code: https://play.golang.org/p/u6b3qTr1CH
video: 082

Reading documentation

It is good to become familiar with the language used to talk about this language. The “golang spec” can be difficult to read. **I like spending some time with my students reading the language specification together so that they gain the skill of being able to read it on their own.**
code:

- https://play.golang.org/p/KkV2YNuQit

video: 083

Anonymous structs

We can create anonymous structs also. **An anonymous struct is a struct which is not associated with a specific identifier.**
code:

- **NAMED**
    - https://play.golang.org/p/O7DiZX_e1R

- **ANONYMOUS**
    - **https://go.dev/play/p/OfqUspH-JIw**

video: 084

Housekeeping

- It's all about type
    - Is go an object oriented language? Go has OOP aspects. But it's all about **TYPE**. We create **TYPES** in Go; user-defined **TYPES**. We can then have **VALUES** of that type. We can assign **VALUES** of a user-defined **TYPE** to **VARIABLES**. Anecdote: makes me think of that song, "It's all about the bass, all about the bass" except "it's all about the **TYPE**, all about the **TYPE**"
- Go is Object Oriented
    1. **Encapsulation**
        a. state ("fields")
        b. behavior ("methods")
        c. exported & unexported; viewable & not viewable
    2. **Reusability**
        a. inheritance ("embedded types")
    3. **Polymorphism**
        a. interfaces
    4. **Overriding**
        a. "promotion"
- Traditional OOP
    1. Classes
        a. data structure describing a type of object
        b. you can then create "instances"/"objects" from the class / blueprint
        c. classes hold both:
            i. state / data / fields
            ii. behavior / methods
        d. public / private
    2. Inheritance
- In Go:
    1. you don't create classes, you create a **TYPE**
    2. you don't instantiate, you create a **VALUE** of a **TYPE**
- User defined types
    - We can declare a new type
    - foo
        - the underlying type of foo is int
        - int conversion
            - int(myAge)
            - converting type foo to type int
    - IT IS A BAD PRACTICE TO ALIAS TYPES

        - one exception:
            - if you need to attach methods to a type
            - see the time package for an example of this godoc.org/time
                - type Duration int64
                - Duration has methods attached to it
- Named types vs anonymous types
        - Anonymous types are indeterminate. They have not been declared as a type yet. The compiler has flexibility with anonymous types. You can assign an anonymous type to a variable declared as a certain type. If the assignment can occur, the compiler will figure it out; the compiler will do an implicit conversion. You cannot assign a named type to a different named type.
- Padding & architectural alignment
        - Convention: logically organize your fields together. Readability & clarity trump performance as a design concern. Go will be performant. Go for readability first. However, if you are in a situation where you need to prioritize performance: lay the fields out from largest to smallest, eg, int 64, int64, float32, bool

code:

- **https://play.golang.org/p/eQP7I3dW_r**

video: 085

Exercises - Ninja Level 5

Hands-on exercise #1

Create your own type “person” which will have an underlying type of “struct” so that it can store the following data:

- first name
- last name
- favorite ice cream flavors

Create two VALUES of TYPE person. Print out the values, ranging over the elements in the slice which stores the favorite flavors.

solution:

- **https://play.golang.org/p/ouRHmH_POg**

video: 086

Hands-on exercise #2

Take the code from the previous exercise, then store the values of type person in a map with the key of last name. Access each value in the map. Print out the values, ranging over the slice.

solution: https://play.golang.org/p/RvvLyAMvGo
video: 087

Hands-on exercise #3

- Create a new type: vehicle.
    - The underlying type is a struct.
    - The fields:
        - doors
        - color
- Create two new types: truck & sedan.
    - The underlying type of each of these new types is a struct.
    - Embed the “vehicle” type in both truck & sedan.
    - Give truck the field “fourWheel” which will be set to bool.
    - Give sedan the field “luxury” which will be set to bool. solution
- Using the vehicle, truck, and sedan structs:
    - using a composite literal, create a value of type truck and assign values to the fields;
    - using a composite literal, create a value of type sedan and assign values to the fields.
- Print out each of these values.
- Print out a single field from each of these values.

solution: https://play.golang.org/p/PrTtTv_vVO
video: 088

Hands-on exercise #4

Create and use an anonymous struct.
solution: https://play.golang.org/p/otBHFs-lPp
video: 089

Functions

Syntax

func (receiver) identifier(parameters) (returns) { code }
know the difference between parameters and arguments

- we define our func with **parameters** (if any)
- we call our func and pass in **arguments** (in any)

Everything in Go is PASS BY VALUE
purpose of functions

- abstract code
- code reusability

code:

- **basic func**
  - https://play.golang.org/p/Ou7esJnAkv
- **takes an argument**
  - https://play.golang.org/p/dplNmrlSsQ
- **return**
  - https://play.golang.org/p/eh2Aud2jyr
- **multiple returns**
  - https://play.golang.org/p/7Xl9uVH2pO

video: 090

Variadic parameter

You can create **a func which takes an unlimited number of arguments.** When you do this, this is known as a "variadic parameter." When use the lexical element operator "...T" to signify a variadic parameter (there "T" represents some type).

code:

- https://play.golang.org/p/Yi0FsQ2tKq

video: 091

Unfurling a slice

When you have a slice of some type, you can **pass in the individual values in a slice by using the "..." operator.**

code:

- **unfurling a slice of int**
  - https://play.golang.org/p/T-mm6-SH71
- **passing in zero or more values**
  - https://play.golang.org/p/Qc5sq_AK_T
- **variadic parameter has to be the final parameter**
  - https://play.golang.org/p/euQ8PDQ8RN

video: 092

Defer

- A "defer" statement invokes a function whose **execution is deferred to the moment the surrounding function returns**, either because the surrounding function executed a return statement, reached the end of its function body, or because the corresponding goroutine is panicking.
- personal anecdote: head down, ox plowing field; doing the work

code: https://play.golang.org/p/AYY3AN4LQ6

video: 093

Methods

A method is nothing more than a FUNC attached to a TYPE. When you attach a func to a type it is a method of that type. You attach a func to a type with a RECEIVER.
code: https://play.golang.org/p/HIBt2HHiIm
video: 094

Interfaces & polymorphism

In Go, values can be of more than one type. An interface allows a value to be of more than one type. We create an interface using this syntax: “keyword identifier type” so for an interface it would be: “type human interface” We then define which method(s) a type must have to implement that interface. If a TYPE has the required methods, which could be none (the empty interface denoted by interface{}), then that TYPE *implicitly implements* the interface and is ***also*** of that interface type. In Go, values can be of more than one type.
code: https://play.golang.org/p/rZH2Efbpot
video: 095

Anonymous func

Anonymous self-executing func
code: https://play.golang.org/p/54U7XWrNwZ
video: 096

func expression

Assigning a func to a variable
code: https://play.golang.org/p/2ekrbY9SAm
video: 097

Returning a func

You can return a func from a func. Here is what that looks like.
code:

- **returning a string**
  - https://play.golang.org/p/w3S9B1Vtyx
- **returning a func**
  - **step 1:** https://play.golang.org/p/3Xk0wLFre8
  - **step 2 - cleaned up:** https://play.golang.org/p/NocmYezUP2
  - **step 3 - cleaned up:** https://play.golang.org/p/FSjvOfY0wW
  - **step 4 - cleaned up:** https://play.golang.org/p/7wbv9KNIhK
  - **step 5 - cleaned up:** https://play.golang.org/p/vW0lGelAox

video: 098

Callback

- **passing a func as an argument**
- functional programming not something that is recommended in go, however, it is good to be aware of callbacks
- idiomatic go: write clear, simple, readable code

code:

- **pre lecture prep: https://play.golang.org/p/j6lXDY_6H2**
- **SIMPLE:**
  - **https://play.golang.org/p/2CJ1TYzv37b**
  - **https://play.golang.org/p/FYtZUVo1yqW**
- **lecture:**
  - **math operators: https://play.golang.org/p/sTDJ3l_rlj**
  - **just sum func: https://play.golang.org/p/TEZChnAYIq**
  - **even numbers: https://play.golang.org/p/RKHjy9Bl6j**
  - **odd numbers: https://play.golang.org/p/Nf3_KrpidO**

video: 099

Closure

- one scope enclosing other scopes
  - variables declared in the outer scope are accessible in inner scopes
- **closure helps us limit the scope of variables**

code:

- **scope of x: https://play.golang.org/p/YWuniJtu2R**
- **scope of x narrowed to func main: https://play.golang.org/p/4hqrzybcFc**
- **code block in code block with y: https://play.golang.org/p/6Hyqe_aU-R**
- **enclosing a variable in a block of code: https://play.golang.org/p/fHez3lg8wc**

video: 100

Recursion

- **a func that calls itself**
- factorial example

code:

- **factorial with recursion: https://play.golang.org/p/fd9nXrqEGi**
- **factorial with loop: https://play.golang.org/p/-GOTqkEUcY**

video: 101

Exercises - Ninja Level 6

Hands-on exercise #1

- Review
    - functions
    - purpose of functions
        - abstract code
        - code reusability
            - DRY - Don't Repeat Yourself
    - func, receiver, identifier, params, returns
    - parameters vs arguments
    - variadic funcs
        - multiple "variadic" params
        - multiple "variadic" args
    - returns
        - multiple returns
        - named returns - yuck!
    - func expressions
        - assigning a func to a variable
    - callbacks
        - passing a func into another func as an argument
    - closure
        - one scope enclosing another
        - variables declared in the outer scope are accessible in inner scopes
        - closure helps us limit the scope of variables
    - recursion
        - a func that calls itself
        - factorial
- life philosophy
    - focus on what's important; not upon what's urgent
- Hands on exercise
    - create a func with the identifier foo that returns an int
    - create a func with the identifier bar that returns an int and a string
    - call both funcs
    - print out their results

code: https://play.golang.org/p/owEJNF5WAD
video: 102

Hands-on exercise #2

- create a func with the identifier foo that
    - takes in a variadic parameter of type int
    - pass in a value of type []int into your func (unfurl the []int)
    - returns the sum of all values of type int passed in
- create a func with the identifier bar that
    - takes in a parameter of type []int
    - returns the sum of all values of type int passed in

code: https://play.golang.org/p/B0yRxtBQPD
video: 103

Hands-on exercise #3

Use the “defer” keyword to show that a deferred func runs after the func containing it exits.
code: https://play.golang.org/p/XluEuUD0Nw
video: 104

Hands-on exercise #4

- Create a user defined struct with
    - the identifier “person”
    - the fields:
        - first
        - last
        - age
- attach a method to type person with
    - the identifier “speak”
    - the method should have the person say their name and age
- create a value of type person
- call the method from the value of type person

code: https://play.golang.org/p/NnXyWdqbbw
video: 105

Hands-on exercise #5

- create a type **SQUARE**
- create a type **CIRCLE**
- attach a method to each that calculates **AREA** and returns it
    - circle area= π r 2
    - square area = L * W
- create a type **SHAPE** that defines an interface as anything that has the **AREA** method
- create a func **INFO** which takes type shape and then prints the area

- create a value of type square
- create a value of type circle
- use func info to print the area of square
- use func info to print the area of circle

code: https://play.golang.org/p/NGGikHNvHv
video: 106

Hands-on exercise #6

- Build and use an anonymous func

code: https://play.golang.org/p/DQX3xEIcRe
video: 107

Hands-on exercise #7

- Assign a func to a variable, then call that func

code: https://play.golang.org/p/_Qu7ZAyFDH
video: 108

Hands-on exercise #8

- Create a func which returns a func
- assign the returned func to a variable
- call the returned func

code: https://play.golang.org/p/c2HwqVE1Rd
video: 109

Hands-on exercise #9

A “callback” is when we pass a func into a func as an argument. For this exercise,

- pass a func into a func as an argument

code: https://play.golang.org/p/0yGYPKh1y7
video: 110

Hands-on exercise #10

Closure is when we have “enclosed” the scope of a variable in some code block. For this hands-on exercise, create a func which “encloses” the scope of a variable:

code: https://play.golang.org/p/a56uWtoFSL
video: 111

Pointers

What are pointers?

All values are stored in memory. Every location in memory has an address. **A pointer is a memory address.**

&
*int
*

code

- **code from video: https://play.golang.org/p/Ysv5Adn3V1**
- **step 1 - take an address & : https://play.golang.org/p/BO7zRQN4Xm**
- **step 2 - dereference * : https://play.golang.org/p/ucJYPu3QkP**
- **step 3 - dereference * : https://play.golang.org/p/KpLImTmQCa**

video: 113

When to use pointers

Pointers allow you to share a value stored in some memory location. Use pointers when

1. you don't want to pass around a lot of data
2. you want to change the data at a location

Everything in Go is pass by value. Drop any phrases and concepts you may have from other languages. Pass by reference, pass by copy - forget those phrases. "Pass by value." That is the only phrase you need to know and remember. That is the only phrase you should use. Pass by value. Everything in Go is pass by value. In Go, what you see is what you get (wysiwyg). Look at what is happening. That is what you get.

code:

- **step 1 no pointer: https://play.golang.org/p/IxsWkhTaYv**
- **step 2 pointer: https://play.golang.org/p/XuI19kjFmb**

video: 114

Method sets

Method sets determine what methods attach to a TYPE. It is exactly what the name says: **What is the set of methods for a given type? That is its method set.**

IMPORTANT: "The method set of a type determines the INTERFACES that the type implements....."

~ golang spec

The above “important” was left out of this video but will be discussed in the “concurrency” section in a video called “method sets revisited”.

- a NON-POINTER RECEIVER
    - works with values that are POINTERS or NON-POINTERS.
- a POINTER RECEIVER
    - only works with values that are POINTERS.

Receivers	Values
(t T)	T and *T
(t *T)	*T

code:

- **NON-POINTER RECEIVER & NON-POINTER VALUE**
    - **https://play.golang.org/p/2ZU0QX12a8**
- **NON-POINTER RECEIVER & POINTER VALUE**
    - **https://play.golang.org/p/glWZmm0gY6**
- **POINTER RECEIVER & POINTER VALUE**
    - **https://play.golang.org/p/pWFxsg6MSe**
- **POINTER RECEIVER & NON-POINTER VALUE**
    - **https://play.golang.org/p/G3lEy-4Mc8 (code does not run)**
    - **this codes does run - notice the difference - method set determines the INTERFACES that the type implements**
        - **https://play.golang.org/p/KK8gjsAWBZ**

video: 115

Exercises - Ninja Level 7

Hands-on exercise #1

- Create a value and assign it to a variable.
- Print the address of that value.

code: https://play.golang.org/p/57kW8xd0qT
video: 116

Hands-on exercise #2

- create a person struct
- create a func called “changeMe” with a *person as a parameter
    - change the value stored at the *person address
        - **important**

        - to dereference a struct, use (*value).field
            - p1.first
            - (*p1).first
        - “As an exception, if the type of x is a named pointer type and (*x).f is a valid selector expression denoting a field (but not a method), x.f is shorthand for (*x).f.”
            - https://golang.org/ref/spec#Selectors
- create a value of type person
    - print out the value
- in func main
    - call “changeMe”
- in func main
    - print out the value

code: https://play.golang.org/p/AehM662HKS
video: 117

Application

JSON documentation

We understand pointers; we understand methods. We now have enough knowledge to begin using the standard library. This video will give you **an orientation to how I approach, read, and work with the standard library. I made a big assumption here that JSON and marshalling would be understood. If this is new to you, read these notes from a fellow student (https://docs.google.com/document/d/1gqTe1ouyvGXKaD1kBBN9hNbTaDBSGZ6Ynt3GBFUbtM0/edit?usp=sharing) .**
video: 118

JSON marshal

Here is an example of how you **marshal data in Go to JSON.** Also important, this video shows how the case of an identifier - lowercase or uppercase, determines whether or not the data can be exported.
code:

- **https://play.golang.org/p/jtE_n16cQO**

video: 119

JSON unmarshal

We can take JSON and bring it back into our Go program by unmarshalling that JSON.
Great resources for understanding and working with JSON are shared.
cool websites:

- http://rawgit.com/
- https://mholt.github.io/json-to-go/
- https://github.com/goestoeleven

code:

- **understanding JSON rawgit HTML page**
- **https://play.golang.org/p/O9q0DmpzsZ**

video: 120

Writer interface

Understanding **the writer interface from package io**. Also, one last note about working with JSON: encode & decode.

code: https://play.golang.org/p/3Txh-dKQBf

video: 121

Sort

The sort package allows us to sort slices.

code:

- **starting code:**
  - **https://play.golang.org/p/iglGnMv6AN**
- **sorted**
  - **https://play.golang.org/p/8UkvEdzQOk**

video: 122

Sort custom

Here is how we **sort on fields in a struct.**

code:

- **starting code:**
  - **https://play.golang.org/p/UhXN-G2FwY**
- **sorted**
  - **by age: https://play.golang.org/p/kqmJovOU5V**
  - **by name: https://play.golang.org/p/he70VcFmdM**

video: 123

bcrypt

All too often today you still hear about websites and databases being hacked and user's information being compromised. There is no excuse for this. As programmers, we have the tools to protect user data. Bcrypt is one of the tools you can use to protect user data. **Using bcrypt, we will gain further understanding as to how to read and implement code from the standard library.**

code:

- **https://goo.gl/ZVKnRx**

video: 124

Exercises - Ninja Level 8

Hands-on exercise #1

Starting with this code, marshal the []user to JSON. There is a little bit of a curve ball in this hands-on exercise - remember to ask yourself what you need to do to EXPORT a value from a package.

solution: https://play.golang.org/p/8BK6PXj3aG

video: 125

Hands-on exercise #2

Starting with this code, unmarshal the JSON into a Go data structure. Hint: https://mholt.github.io/json-to-go/

code:

- **code setup (just fyi, not needed for exercise):**
  - **https://play.golang.org/p/nWPP5Z2Q4e**
- **solution:**
  - **https://play.golang.org/p/r8oSG8DlPR**

video: 126

Hands-on exercise #3

Starting with this code, encode to JSON the []user sending the results to Stdout. Hint: you will need to use json.NewEncoder(os.Stdout).encode(v interface{})

solution: https://play.golang.org/p/ql90D1OQqd

video: 127

Hands-on exercise #4

Starting with this code, sort the []int and []string for each person.

solution: https://play.golang.org/p/jz_lIY1aPp

video: 128

Hands-on exercise #5

Starting with this code, sort the []user by

- age
- last

Also sort each []string “Sayings” for each user

- print everything in a way that is pleasant

solution: https://play.golang.org/p/8RKkdtLl6w
video: 129

Concurrency

Concurrency vs parallelism

But when people hear the word *concurrency* they often think of *parallelism*, a related but quite distinct concept. In programming, concurrency is the *composition* of independently executing processes, while parallelism is the simultaneous *execution* of (possibly related) computations. Concurrency is about *dealing with* lots of things at once. Parallelism is about *doing* lots of things at once.

- First Intel dual-core processor: 2006. Google begins developing a language to take advantage of multiple cores: 2007

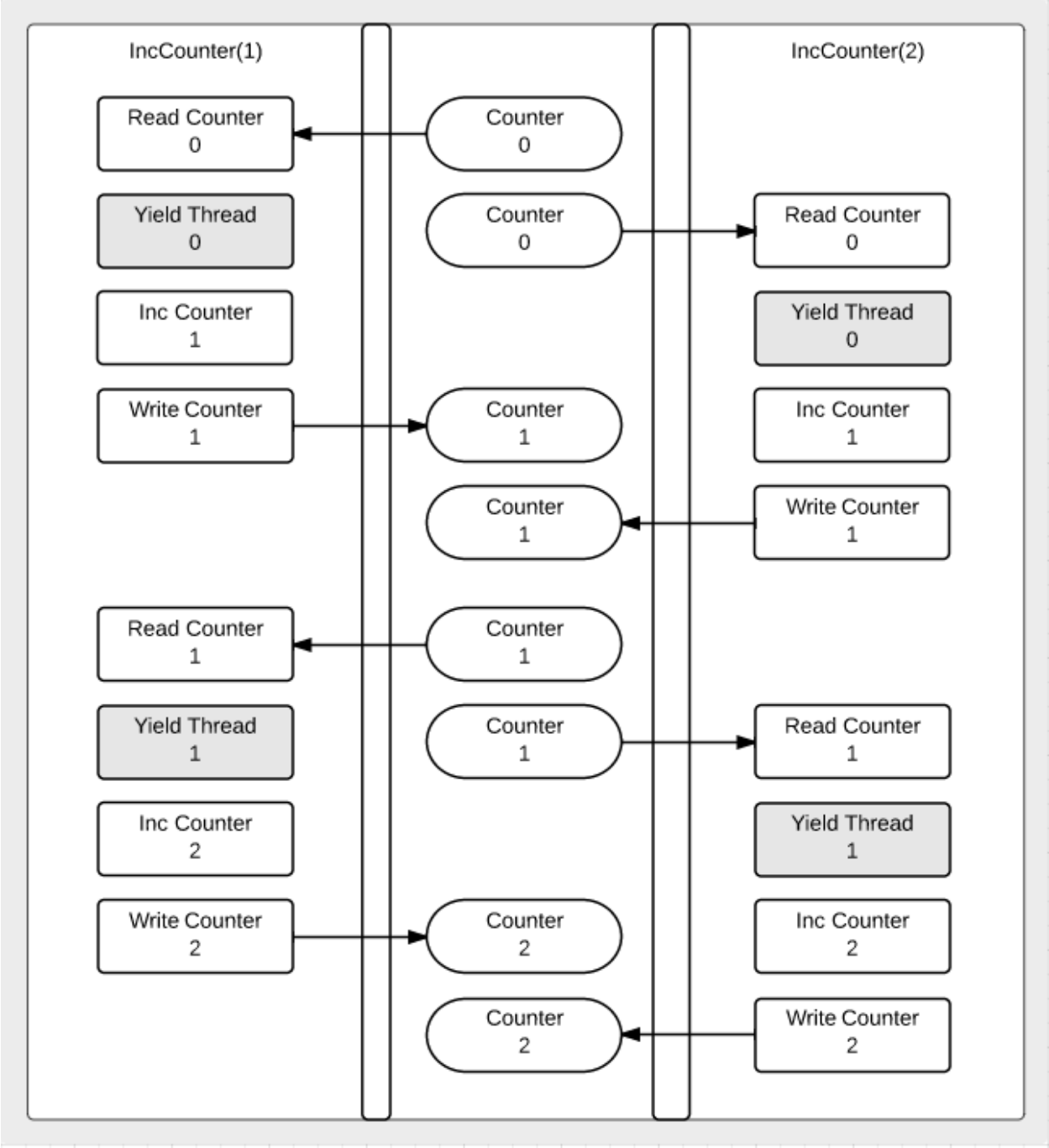
video: 130

WaitGroup

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished. Writing concurrent code is super easy: all we do is **put “go” in front of a function or method call**.

- runtime.NumCPU()
- runtime.NumGoroutine()
- sync.WaitGroup
    - func (wg *WaitGroup) Add(delta int)
    - func (wg *WaitGroup) Done()
    - func (wg *WaitGroup) Wait()

race conditions picture:



- **file can also be found in the MISCELLANEOUS RESOURCES lecture on UDEMY**

code:

- **starting code:**
  - **https://play.golang.org/p/bnl0akWF9f**
- **finished:**
  - **https://play.golang.org/p/VDioqpZ65h**

video: 131

Method sets revisited

“**The method set of a type determines the INTERFACES that the type implements**.....” ~ golang spec

Receivers	Values
(t T)	T and *T
(t *T)	*T

code: https://play.golang.org/p/KK8gjsAWBZ
video: 132

Documentation

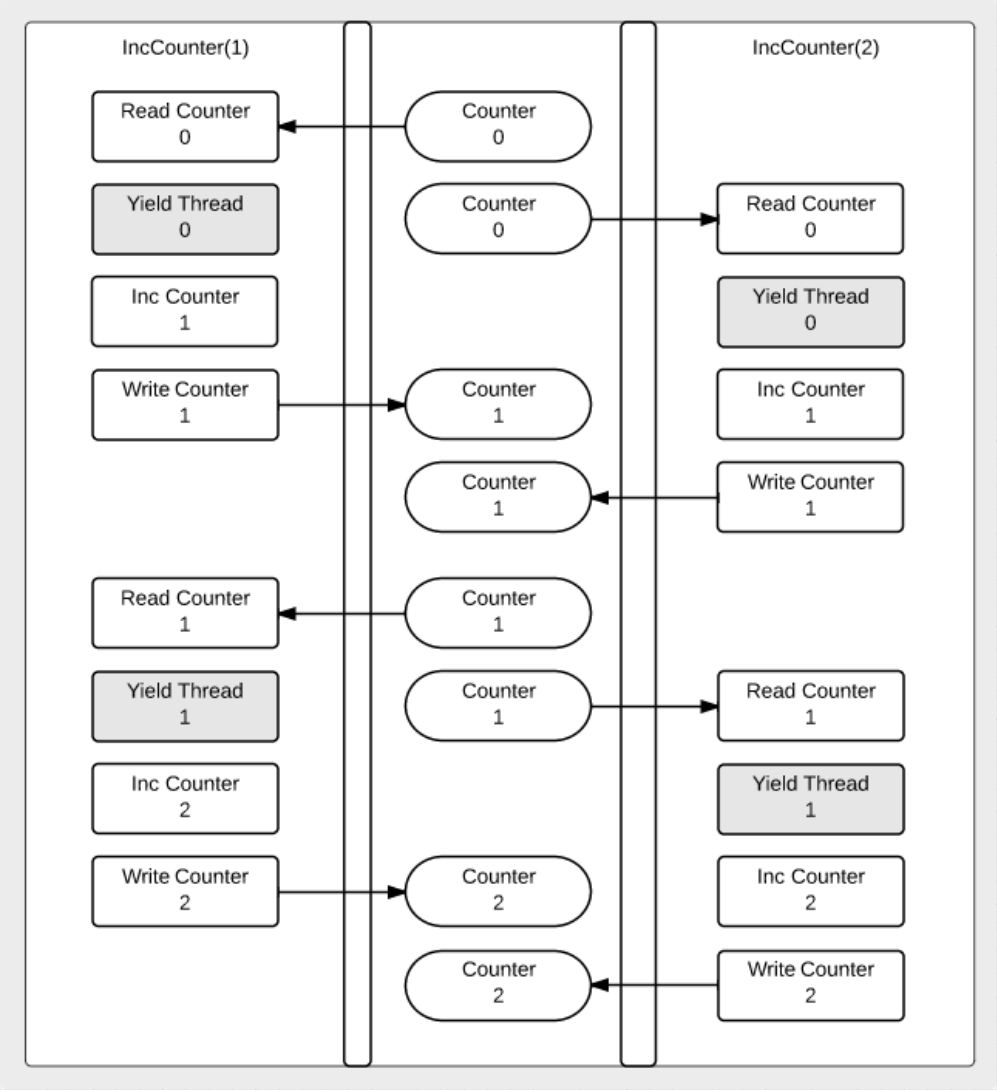
They're called goroutines because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space.
code:

- **https://play.golang.org/p/lBKFKCwrue**

video: 133

Race condition

Here is a picture of the race condition we are going to create:



Race conditions are not good code. A race condition will give unpredictable results. We will see how to fix this race condition in the next video.
code: https://play.golang.org/p/FYGofIKQej
video: 134

Mutex

A “mutex” is a mutual exclusion lock. **Mutexes allow us to lock our code so that only one goroutine can access that locked chunk of code at a time.**
code: https://github.com/GoesToEleven/golang-web-dev/tree/master/000_temp/52-race-condition
video: 135

Atomic

We can **use package atomic to also prevent a race condition** in our incrementer code.
code: https://github.com/GoesToEleven/golang-web-dev/tree/master/000_temp/52-race-condition

video: 136

Exercises - Ninja Level 9

Hands-on exercise #1

- in addition to the main goroutine, launch two additional goroutines
    - each additional goroutine should print something out
- use waitgroups to make sure each goroutine finishes before your program exists

code: https://github.com/GoesToEleven/go-programming
video: 148

Hands-on exercise #2

This exercise will reinforce our understanding of method sets:

- create a type person struct
- attach a method speak to type person using a pointer receiver
    - *person
- create a type human interface
    - to implicitly implement the interface, a human must have the speak method
- create func “saySomething”
    - have it take in a human as a parameter
    - have it call the speak method
- show the following in your code
    - you CAN pass a value of type *person into saySomething
    - you CANNOT pass a value of type person into saySomething
- here is a hint if you need some help
    - https://play.golang.org/p/FAwcQbNtMG

Receivers Values

(t T) T and *T
(t *T) *T

code: https://github.com/GoesToEleven/go-programming
video: 149

Hands-on exercise #3

- Using goroutines, create an incrementer program
    - have a variable to hold the incrementer value
    - launch a bunch of goroutines
        - each goroutine should

        - read the incrementer value
            - store it in a new variable
        - yield the processor with runtime.Gosched()
        - increment the new variable
        - write the value in the new variable back to the incrementer variable
- use waitgroups to wait for all of your goroutines to finish
- the above will create a race condition.
- Prove that it is a race condition by using the -race flag
- if you need help, here is a hint: https://play.golang.org/p/FYGoflKQej

code: https://github.com/GoesToEleven/go-programming
video: 150

Hands-on exercise #4

Fix the race condition you created in the previous exercise by using a mutex

- it makes sense to remove runtime.Gosched()

code: https://github.com/GoesToEleven/go-programming
video: 151

Hands-on exercise #5

Fix the race condition you created in exercise #4 by using package atomic
code: https://github.com/GoesToEleven/go-programming
video: 152

Hands-on exercise #6

Create a program that prints out your OS and ARCH. Use the following commands to run it

- go run
- go build
- go install

code: https://github.com/GoesToEleven/go-programming
video: 153

Channels

Understanding channels

Channels Introduction

- making a channel

```
c := make(chan int)
```

- putting values on a channel

c <- 42

- taking values off of a channel

  <-c

- buffered channels

  c := make(chan int, 4)

- channels block
  - they are like runners in a relay race
    - they are synchronized
    - they have to pass/receive the value at the same time
      - just like runners in a relay race have to pass / receive the baton to each other at the same time
        - one runner can't pass the baton at one moment
        - and then, later, have the other runner receive the baton
        - the baton is passed/received by the runners at the same time
      - the value is passed/received synchronously; at the same time
- channels allow us to pass values between goroutines

code:

- **doesn't work**
  - **https://play.golang.org/p/XPgsj2xS0F**
  - **IMPORTANT: CHANNELS BLOCK**
    - channels allow
      - coordination / synchronization / orchestration
      - buffering (buffered channels)
- **send & receive**
  - **https://play.golang.org/p/SHr3lpX4so**
- **buffer**
  - **https://play.golang.org/p/hsttb2qEJi**
    - *"The capacity, in number of elements, sets the size of the buffer in the channel. If the capacity is zero or absent, the channel is unbuffered and communication succeeds only when both a sender and receiver are ready."* *Golang Spec*
- **buffer doesn't work**
  - **https://play.golang.org/p/epLsvcivJS**
- **buffer**
  - **https://play.golang.org/p/_6bSl5fc17**

code: https://github.com/GoesToEleven/go-programming
video: 155

Directional channels

Channel type. Read from left to right.

code:

- **seeing the type**
    - **code from previous video**
        - **https://play.golang.org/p/a98otBr4eX**
    - **send & receive (bidirectional)**
        - **https://play.golang.org/p/di1mKkGF6Y**
        - **"send and receive" means "send and receive"**
            - **https://play.golang.org/p/SHr3lpX4so**
                - already saw the above code
        - **send means send**
            - **error:** *"invalid operation: <-cs (receive from send-only type chan<- int)"*
                - **https://play.golang.org/p/oB-p3KMiH6**
        - **receive means receive**
            - **error:** *"invalid operation: cr <- 42 (send to receive-only type <-chan int)"*
                - **https://play.golang.org/p/_DBRuelmEq**
        - *"A channel may be constrained only to send or only to receive [general to specific] by conversion or assignment."* *Golang Spec*
            - doesn't assign
                - specific to general
                    - https://play.golang.org/p/bG7H6l03VQ
                - specific to specific
                    - https://play.golang.org/p/8JkOnEi7-a
            - assigns
                - general to specific
                    - https://play.golang.org/p/hrNZq961KA
            - conversion
                - general to specific works
                    - https://play.golang.org/p/H1uk4YGMBB
                - specific to general doesn't work
                    - https://play.golang.org/p/4sOKuQRHq7

code: https://github.com/GoesToEleven/go-programming

video: 156

Using channels

- create general channels
- in funcs you can specify
    - receive channel
        - you can receive values from the channel
        - a receive channel parameter

        - in the func, you can only pull values from the channel
        - you can't close a receive channel
    - send channel
        - you can push values to the channel
        - you can't receive/pull/read from the channel
        - you can only push values to the channel

code:

- **https://play.golang.org/p/t1rc8rSrMd**

code: https://github.com/GoesToEleven/go-programming
video: 157

Range

Range stops reading from a channel when the channel is closed.

code:

- **close a channel**
    - **https://play.golang.org/p/w4KMBpSEyj**
- **range over a channel**
    - **https://play.golang.org/p/tUVjK5QSQB**

code: https://github.com/GoesToEleven/go-programming
video: 158

Select

Select statements pull the value from whatever channel has a value ready to be pulled.

code:

- **building on previous code**
    - **https://play.golang.org/p/41m5Mt7B-5**
- **not closing even odd channels**
    - **https://play.golang.org/p/Emx6S4zn_y**

code: https://github.com/GoesToEleven/go-programming
video: 159

Comma ok idiom

The comma ok idiom with select.

code:

- **closing quit channel & comma ok idiom**
    - **https://play.golang.org/p/fomc4Sc-gz**
    - **with bool**
        - **https://play.golang.org/p/QAwBLDKZuq**
    - **with int**
        - **https://play.golang.org/p/6XaTWxKpU3**

- **just comma ok idiom**
    - **https://play.golang.org/p/6LPzCtZeT3**
    - **https://play.golang.org/p/dToDc0zJhZ**
- **clean up above code - comma ok idiom**
    - **step 1 - comma ok idiom code reduced**
        - **https://play.golang.org/p/60K5-7xat6**
    - **step 2 - remove underscore variable throwaways**
        - **https://play.golang.org/p/QWzGUlSkJK**
    - **step 3 - change quit from bool to int**
        - **https://play.golang.org/p/tFlvGg9ENT**
- **select receive**
    - **https://play.golang.org/p/62vRH3jeQ6**
- **select send**
    - **https://play.golang.org/p/bv_vCcJ6zs**
- **interesting**
    - **doesn't run**
        - **https://play.golang.org/p/o7Quy6wc6r**
    - **runs**
        - **https://play.golang.org/p/8ZMDPBqHwt**
    - **runs a different way - I like this one better**
        - **quit channel was removed**
        - **select was removed**
            - **range over channel used instead**
        - **https://play.golang.org/p/_CyyXQBCHe**

video: 160

Fan in

Taking values from many channels, and putting those values onto one channel.

code:

- **Todd's code**
    - **https://play.golang.org/p/_CyyXQBCHe**
- **Rob Pike's code**
    - **https://play.golang.org/p/buy30qw5MM**

video: 161

Fan out

Taking some work and putting the chunks of work onto many goroutines.

code:

- **fan out in**
    - **https://play.golang.org/p/iU7Oee2nm7**
- **throttle throughput**

- **https://play.golang.org/p/RzR3Kjrx7q**

video: 162

Context

In Go servers, each incoming request is handled in its own goroutine. Request handlers often start additional goroutines to access backends such as databases and RPC services. The set of goroutines working on a request typically needs access to request-specific values such as the identity of the end user, authorization tokens, and the request's deadline. When a request is canceled or times out, all the goroutines working on that request should exit quickly so the system can reclaim any resources they are using. At Google, we developed a context package that makes it easy to pass request-scoped values, cancellation signals, and deadlines across API boundaries to all the goroutines involved in handling a request. The package is publicly available as context. This article describes how to use the package and provides a complete working example.

further reading:

- **https://blog.golang.org/context**
- **https://medium.com/@matryer/context-has-arrived-per-request-state-in-go-1-7-4d095be83bd8**
- **https://peter.bourgon.org/blog/2016/07/11/context.html**

code:

- exploring context
  - background
    - https://play.golang.org/p/cByXyrxXUf
  - WithCancel
    - throwing away CancelFunc
      - https://play.golang.org/p/XOknf0aSpx
    - using CancelFunc
      - https://play.golang.org/p/UzQxxhn_fm
    - Example
      - **https://play.golang.org/p/Lmbyn7bO7e**
- `func WithCancel(parent Context) (ctx Context, cancel CancelFunc)`
  - **https://play.golang.org/p/wvGmvMzIMW**
- cancelling goroutines with deadline
- `func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)`
  - https://play.golang.org/p/Q6mVdQqYTt
- with timeout
- `func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)`
  - https://play.golang.org/p/OuES9sP_yX
- with value
- `func WithValue(parent Context, key, val interface{}) Context`
  - https://play.golang.org/p/8JDCGk1K4P

video: 163

Exercises - Ninja Level 10

Hands-on exercise #1

- get this code working:
    - using func literal, aka, anonymous self-executing func
        - solution: https://play.golang.org/p/SHr3lpX4so
    - a buffered channel
        - solution: https://play.golang.org/p/Y0Hx6lZc3U

video: 164

Hands-on exercise #2

- Get this code running:
    - **https://play.golang.org/p/oB-p3KMiH6**
        - **solution: https://play.golang.org/p/isnJ8hMMKg**
    - **https://play.golang.org/p/_DBRueImEq**
        - **solution: https://play.golang.org/p/mgw750EPp4**

video: 165

Hands-on exercise #3

- Starting with this code, pull the values off the channel using a for range loop

solution: https://play.golang.org/p/D3N4Tq54SN
video: 166

Hands-on exercise #4

- Starting with this code, pull the values off the channel using a select statement

solution: https://play.golang.org/p/FulKBY5JNj
video: 167

Hands-on exercise #5

- Show the comma ok idiom starting with this code.

solution: https://play.golang.org/p/qh2ywLB5OG
video: 168

Hands-on exercise #6

- write a program that
    - puts 100 numbers to a channel

        - pull the numbers off the channel and print them

solution: **https://play.golang.org/p/096Lk1BR7o**
video: 169

Hands-on exercise #7

- write a program that
    - launches 10 goroutines
        - each goroutine adds 10 numbers to a channel
    - pull the numbers off the channel and print them

solutions:

- **https://play.golang.org/p/R-zqsKS03P**
- **https://play.golang.org/p/quWnlwzs2z**
- **https://go.dev/play/p/WqYnBC_CiKn**

video: 170

Error handling

Understanding

- https://golang.org/doc/faq#exceptions
    - Go does not favor try / catch / finally
- https://en.wikipedia.org/wiki/Exception_handling#Criticism
    - Notice Hoare's work also influenced goroutines and channels
- **https://blog.golang.org/error-handling-and-go**

video: 171

Checking errors

Write the code with errors before writing the code without errors. **Always check for errors. Always, always, always.***
(*almost always)
code:

- **https://github.com/GoesToEleven/go-programming**

video: 172

Printing & logging

You have a few options to choose from when it comes to printing out, or logging, an error message:

- fmt.Println()
- log.Println()
- log.Fatalln()

    - os.Exit()
- log.Panicln()
    - deferred functions run
    - can use “recover”
- panic()

code:

- **https://github.com/GoesToEleven/go-programming**

video: 173

Recover

https://blog.golang.org/defer-panic-and-recover

code:

- **https://play.golang.org/p/Hl4uG55ait**
- **https://play.golang.org/p/ZocncqtwaK**

video: 174

Errors with info

We can add information to our errors. We can do this with

- errors.New()
    - fmt.Errorf()
- builtin.error

“Error values in Go aren’t special, they are just values like any other, and so you have the entire language at your disposal.” - Rob Pike

code:

- **https://github.com/GoesToEleven/go-programming**

video: 175

Exercises - Ninja Level 11

Hands-on exercise #1

Start with this code. Instead of using the blank identifier, make sure the code is checking and handling the error.

solution:

- **https://play.golang.org/p/tn8oiuL1Yn**

video: 176

Hands-on exercise #2

Start with this code. Create a custom error message using “fmt.Errorf”.

solution:

- **https://play.golang.org/p/HugU4HJEEO**
- **https://play.golang.org/p/NII-lmGasj**
- **https://play.golang.org/p/Vo5kloR-sG**

video: 177

Hands-on exercise #3

Create a struct “customErr” which implements the builtin.error interface. Create a func “foo” that has a value of type error as a parameter. Create a value of type “customErr” and pass it into “foo”. If you need a hint, here is one.

solution:

- **https://play.golang.org/p/ixeowY2fd2**
- **assertion**
  - **https://play.golang.org/p/pbI2kCYsM0**
- **conversion**
  - **https://play.golang.org/p/1IdiBdkdzA**

video: 178

Hands-on exercise #4

Starting with this code, use the sqrt.Error struct as a value of type error. If you would like, use these numbers for your

- lat "50.2289 N"
- long "99.4656 W"

solution:

- **https://play.golang.org/p/nsRxbDfkCh**

video: 179

Hands-on exercise #5

We are going to learn about testing next. For this exercise, take a moment and see how much you can figure out about testing by reading the testing documentation & also Caleb Doxsey’s article on testing. See if you can get a basic example of testing working.

video: 180

Writing documentation

Introduction

Before writing documentation, we are going to look at reading documentation. There are several things to know about documentation:

- godoc.org
  - standard library and third party package documentation

- golang.org
    - standard library documentation
- go doc
    - command to read documentation at the command line
- godoc
    - command to read documentation at the command line
    - also can run a local server showing documentation

Personal update on my health, mortality, and resources that have helped me cultivate a more skillful mindset in my life:

- Never Split The Difference by Chris Voss
- Grit: The Power of Passion and Perseverance by Angela Duckworth
- Smarter Faster Better: The Secrets of Being Productive in Life and Business
- https://www.entrepreneur.com/topic/masters-of-scale
- http://dharmaseed.org/teachers/

video: 180-a

go doc

go doc prints the documentation for a package, const, func, type, var, or method

- go doc accepts zero, one, or two **arguments**.
    - **zero**
        - prints package documentation for the package in the current directory
            - **go doc**
    - **one**
        - argument Go-syntax-like representation of item to be documented
            - fyi: <sym> also known as “identifier”
                - **go doc <pkg>**
                - **go doc <sym>[.<method>]**
                - **go doc [<pkg>.]<sym>[.<method>]**
                - **go doc [<pkg>.][<sym>.]<method>**
            - The first item in this list that succeeds is the one whose documentation is printed. If there is a symbol but no package, the package in the current directory is chosen. However, if the argument begins with a capital letter it is always assumed to be a symbol in the current directory.
    - **two**
        - first argument must be a full package path
            - **go doc <pkg> <sym>[.<method>]**
- examples

```
Examples:
        go doc
                Show documentation for current package.
        go doc Foo
                Show documentation for Foo in the current package.
                (Foo starts with a capital letter so it cannot match
                a package path.)
        go doc encoding/json
                Show documentation for the encoding/json package.
        go doc json
                Shorthand for encoding/json.
        go doc json.Number (or go doc json.number)
                Show documentation and method summary for json.Number.
        go doc json.Number.Int64 (or go doc json.number.int64)
                Show documentation for json.Number's Int64 method.
        go doc cmd/doc
                Show package docs for the doc command.
        go doc -cmd cmd/doc
                Show package docs and exported symbols within the doc command.
        go doc template.new
                Show documentation for html/template's New function.
                (html/template is lexically before text/template)
        go doc text/template.new # One argument
                Show documentation for text/template's New function.
        go doc text/template new # Two arguments
                Show documentation for text/template's New function.
```

```
At least in the current tree, these invocations all print the
documentation for json.Decoder's Decode method:

go doc json.Decoder.Decode
go doc json.decoder.decode
go doc json.decode
cd go/src/encoding/json; go doc decode
```

video: 180-b

godoc

Godoc extracts and generates documentation for Go programs. It has two modes

- **without -http flag**
  - command-line mode; prints text documentation to standard out and exits
  - **-src** flag
    - godoc prints the exported interface of a package in Go source form, or the implementation of a specific exported language

```
godoc fmt                # documentation for package fmt
godoc fmt Printf         # documentation for fmt.Printf
godoc cmd/go             # force documentation for the go command
godoc -src fmt           # fmt package interface in Go source form
godoc -src fmt Printf    # implementation of fmt.Printf
```

- **with -http flag**
    - runs as a web server and presents the documentation as a web page
    - **godoc -http=:8080**
        - http://localhost:8080/

video: 180-c

godoc.org

- put the url of your code into godoc
    - your documentation will appear on godoc
    - “refresh” at bottom of page if it is ever out of date

video: 180-d

Writing documentation

Documentation is a huge part of making software accessible and maintainable. Of course it must be well-written and accurate, but it also must be easy to write and to maintain. Ideally, it should be coupled to the code itself so the documentation evolves along with the code. The easier it is for programmers to produce good documentation, the better for everyone.

- https://blog.golang.org/godoc-documenting-go-code
    - **godoc** parses Go source code - including comments - and produces documentation as HTML or plain text. The end result is documentation tightly coupled with the code it documents. For example, through godoc's web interface you can **navigate from a function's documentation to its implementation with one click.**
    - comments are just good comments, the sort you would want to read even if **godoc** didn't exist.
    - **to document**
        - a type, variable, constant, function, or package,
        - **write a comment** directly preceding its declaration, with no intervening blank line.
            - **begin with the name of the element**
            - for packages
                - first sentence appears in package list
                - if a large amount of documentation, place in its own file **doc.go**

        - example: package fmt
    - the best thing about godoc's minimal approach is how easy it is to use. As a result, a lot of Go code, including all of the standard library, already follows the conventions.
- example
    - errors package

code: https://github.com/GoesToEleven/go-programming
video: 180-e

Exercises - Ninja Level 12

Hands-on exercise #1

Create a dog package. The dog package should have an exported func "Years" which takes human years and turns them into dog years (1 human year = 7 dog years). Document your code with comments. Use this code in func main.

- run your program and make sure it works
- run a local server with godoc and look at your documentation.

solution: https://github.com/GoesToEleven/go-programming
video: 180-f

Hands-on exercise #2

Push the code to github. Get your documentation on godoc.org and take a screenshot. Delete your code from github. Refresh godoc.org so that it no longer has your code.
solution: https://github.com/GoesToEleven/go-programming
video: no video

Hands-on exercise #3

Use godoc at the command line to look at the documentation for:

- fmt
- fmt Print
- strings
- strconv

solution: https://github.com/GoesToEleven/go-programming
video: no video

Testing & Benchmarking

Introduction

Tests must

- be in a file that ends with "**_test.go**"
- put the file in **the same package** as the one being tested
- be in a func with a signature "**func TestXxx(*testing.T)**"

Run a test

- **go test**

Deal with test failure

- use **t.Error** to signal failure

Nice idiom

- **expected**
- **got**

code: https://github.com/GoesToEleven/go-programming
video: 181

Table tests

We can write **a series of tests to run**. This allows us to test a variety of situations.
code: https://github.com/GoesToEleven/go-programming
video: 182

Example tests

Examples show up in documentation.

- **godoc -http :8080**
- https://blog.golang.org/examples
- **go test ./…**

code: https://github.com/GoesToEleven/go-programming
video: 183

Golint

- **gofmt**
  - formats go code
- **go vet**
  - reports suspicious constructs
- **golint**
  - reports poor coding style

video: 184

Benchmark

Part of the testing package allows us to measure the speed of our code. This could also be called “measuring the performance” of your code, or “benchmarking” your code - finding out how fast the code runs.

code: https://github.com/GoesToEleven/go-programming

video: 185

Coverage

Coverage in programming is how much of our code is covered by tests. We can use the “-cover” flag to run coverage analysis on our code. We can use the flag and required file name “-coverprofile <some file name>” to write our coverage analysis to a file.

code:

- **go test -cover**
    - **go test -coverprofile c.out**
        - **show in browser:**
            - **go tool cover -html=c.out**
        - **learn more**
            - **go tool cover -h**
- **https://github.com/GoesToEleven/go-programming**

video: 186

Benchmark examples

Here are a few examples showing benchmarking in action. This includes comparing manual concatenation with strings.Join

code:

- **https://github.com/GoesToEleven/go-programming**

video: 187

Review

Here is a review of the different commands useful with benchmarks, examples, and tests.

- godoc -http=:8080
- go test
- go test -bench .
    - *don’t forget the “.” in the line above*
- go test -cover
- go test -coverprofile c.out
- go tool cover -html=c.out

code:

- **https://github.com/GoesToEleven/go-programming**

video: 188

Exercises - Ninja Level 13

Hands-on exercise #1

Start with this code. Get the code ready to BET on the code (add benchmarks, examples, tests). Run the following in this order:

- tests
- benchmarks
- coverage
- coverage shown in web browser
- examples shown in documentation in a web browser

solution:

- **https://github.com/GoesToEleven/go-programming**

video: 189

Hands-on exercise #2

Start with this code. Get the code ready to BET on (add benchmarks, examples, tests) ***however*** do not write an example for the func that returns a map; and only write a test for it as an extra challenge. Add documentation to the code. Run the following in this order:

- tests
- benchmarks
- coverage
- coverage shown in web browser
- examples shown in documentation in a web browser

solution:

- **https://github.com/GoesToEleven/go-programming**

video: 190

Hands-on exercise #3

Start with this code. Get the code ready to BET on (add benchmarks, examples, tests). Write a table test. Add documentation to the code. Run the following in this order:

- tests
- benchmarks
- coverage
- coverage shown in web browser
- examples shown in documentation in a web browser

helpful to know:

- **https://play.golang.org/p/4GUqs1HMpp**

- **https://play.golang.org/p/P9unTlFeOq**

solution:

- **https://github.com/GoesToEleven/go-programming**

video: 191

FAREWELL

You have done great work - the greatest work. You have taken steps to create a better life for yourself, and for others. As an individual improves their own life, they improve the world. The skills you are acquiring are some of the most valuable skills demanded today: knowing how to code and knowing how to use the Go programming language.

Next Steps

video: 192

Cross compile

- GOOS & GOARCH
  - http://godoc.org/runtime#pkg-constants
- GOOS=darwin GOARCH=386 go build test.go

video: 146

Packages

- one folder, many files
  - package declaration in every file
  - package scope
    - something in one file is accessible to another file
  - imports have file scope
- exported / unexported
  - aka, visible / not visible
  - we don't say (generally speaking): public / private
  - capitalization
    - capitalize: exported, visible outside the package
    - lowercase: unexported, not visible outside the package

video: 147