Todd McLeod
Learn To Code Go on Udemy - Part 2

TABLE OF CONTENTS

PROMO VIDEO

Video #00
Welcome

INTRODUCTION

Video #01
Resources: accessing github code samples, accessing powerpoint presentations used in my college and university go programming classes

- Resources
- Github code samples
    - downloading zip
    - searching for code
- Powerpoint presentations
- Bill Kennedy's book
- Donovan and Kernighan's book
- Caleb Doxsey's book
- go by example
- language spec
- effective go

Video #02
Why Golang; reasons for choosing go; companies using go;

- Why golang
- Reasons for choosing go
- Companies using go
- People using go (bye bye node.js)
- A chinese company's server load

Video #03
Hello World; basic program structure; the go playground; packages; standard library; golang.org; godoc.org;
code: https://play.golang.org/p/04Qiz0BH4N

- hello world
    - go playground
        - formatting code
        - sharing code
        - run code
- packages

        - code that is already written which you can use
        - imports
            - (importing a package)
- **NOTE:**
    - **golang.org changed to go.dev/**
    - **godoc.org changed to pkg.go.dev/**
- documentation
    - go.dev
        - standard library
        - src
    - pkg.go.dev
        - standard library AND third-party packages

INSTALLING GO

Video #04
Section Overview
a preview of what we're going to cover in this section: overview of the go installation process; gui's versus command line; command prompt vs terminal; DOS commands vs Unix commands; SHA1, hash algorithm checksums; the importance of having your workspace configured correctly; namespacing in go; you will learn how to configure either a windows machine or a mac in this section

Video #05
terminal emulator - github
Download github and change the settings to enable the "git bash" terminal so that we can run Unix terminal commands on a Windows machine.

- download github for windows
- open settings
- choose "git bash"
- open it and test it
- basic terminal commands
    - ls
        - ls -la
    - cd
    - cd ../
    - pwd

Video #06
Installation Insights
download go; run sha1 checksum openssl sha1; uninstall any old versions; install new version

- download go
- sha1 checksum
- uninstall old version
- install new version
- **go version**

Video #07
Go workspace
Learn the importance of your Go workspace. Learn how to configure your Go workspace. Learn about namespacing, the importance of namespacing, and how Go implements namespacing.

- 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

Video #08
Environment Path Variables

- **go env**
- **share this video - preview in courses**

Video #09
Configuring path variables - windows

- computer / properties / advanced system settings / environment variables /
    - googling for location on different windows versions
    - search box in windows
- user variables vs system variables
- GOPATH
    - copying the path from windows explorer
- Setting a PATH variable to your bin
- **share this video - preview in courses**

Video #10

Configuring Path Variables - Mac

- spotlight search
- terminal
- cd ~
    - users home
- ls -la
- hidden files
- .bash_profile
- .bashrc
- nano

Video #11
Testing Your Installation

- Test your installation and, at the same time, get all of the sample code used in this training!
- go get github.com/GoesToEleven/GolangTraining

Video #12
Section Review

- github terminal emulator
- basic terminal commands
    - ls -la
    - cd
    - cd ../
    - pwd
    - env
- sha1 checksum
- go command
    - go
    - go version
    - go env
    - go get
- Go workspace
    - bin
    - pkg
    - src
        - github.com
            - <github.com username>
                - folder with code for project / repo
- GOPATH
- GOROOT
- namespacing
- environment path variables
- using **go get** to get the code used in this course

YOUR DEVELOPMENT ENVIRONMENT

Video #13
Section Overview

- IDE's (integrated development environment)
    - WebStorm
    - Atom.io
- Configuring Webstorm
- Using Webstorm
- go run, go build, go install
- Understanding Github
- Creating Your Own Repo
- Using Github

Video #14
Go Editors

- IDE's
- **Webstorm**
    - plugins
    - golang plugin
    - creating a new project || package || library
    - GO SDK
    - themes
        - http://www.ideacolorthemes.org/home/
    - live edit
        - JETBrains IDE Support Chrome Plugin

Video #15
WebStorm & Atom.io

- keymap
- I have found helpful
    - preferences
        - search (search box in preferences)
        - right margin
        - wrap, soft wraps, folding
        - emmet
        - show line numbers
        - terminal
        - local history
        - refactor / rename
        - find in path

        - reformat code
        - comment
        - surround with
        - VCS
- Atom.io
    - made by github
    - go-plus plugin

Video #16
Creating Your First Project

- creating a folder for your code
- opening that folder with webstorm
- selecting the Go SDK
- organizing your code sequentially
    - naming folders with numbers first is NOT idiomatic
- .idea folder
    - seeing hidden files in Windows

Video #17
Hello World with Webstorm

- writing the program
    - code completion
- seeing documentation and source code internals instantly within webstorm
- terminal
    - go fmt
    - go run
- setting your theme
- **share this video - preview in courses**

Video #18
The Go Command & Documentation

- documentation
- go run
    - needs a file name, eg, go run main.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 the folder name holding the code
        - 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
- go clean

Video #19
Understanding Github

- git
- github
- history
- github
    - code storage
    - code sharing
    - code collaboration
    - code versions
    - code searching
    - programmer assessing

Video #20
Using Github

- windows
    - install git
- Creating Your Own Repo
- pushing code to github
    - from webstorm
    - from terminal
        - git status
        - git add --all
        - git commit -m “first commit”
            - first time - set remote repository (repo)

```
git remote add origin https://github.com/GoesToEleven/udemyTraining.git
git push -u origin master
```

        - git push

Video #21
Section Review

- IDE's
    - webstorm
    - atom
- Creating Your First Project
    - creating a folder for your code
    - opening that folder with webstorm
        - selecting the Go SDK
    - organizing your code sequentially
        - naming folders with numbers first is NOT idiomatic
    - .idea folder
        - seeing hidden files in Windows
- Hello World with Webstorm
    - seeing documentation and source code internals instantly within webstorm
- terminal
    - go fmt
    - go run
    - go build
        - for an executable:
            - builds the file
            - reports errors, if any
            - throws away binary
    - go install
        - for an executable:
            - compiles the program (builds it)
            - names the executable the folder name holding the code
            - puts the executable in **workspace / bin**
                - $GOPATH / bin
- Git & Github
    - code storage, sharing, collaboration
    - code versioning, code searching
    - programmer assessing
- Using Github
    - windows
        - install git
    - Creating Your Own Repo
    - pushing code to github
        - from webstorm
        - from terminal
            - git status
            - git add --all

- git commit -m “first commit”
    - first time - set remote repository (repo)

```
git remote add origin https://github.com/GoesToEleven/udemyTraining.git
git push -u origin master
```

- git push

COMPUTER FUNDAMENTALS

Video #22
Section Overview
This section is going to show you how computers work. You will learn the internal systems of a computer. You will learn about circuits and coding schemes. You will learn about binary, binary digits, and bits. You will learn how to measure bits. You will learn about numeral systems. You will learn how to use binary and hexadecimal. You will learn about format printing and format verbs. You will also learn about the history of computers.

Video #23 - part 1
How Computers Work

- CIRCUITS / SWITCHES
- CODING SCHEMES
- BINARY
- 2n
- 5 generations of computers
- Moore's Law

Video #23 - part 2
How Computers Work

- Measuring Bits
    - Bits
    - Bytes
    - KB
    - MB
    - GB
    - TB
- Machine Language

Video #24
Github Update Command

- Housekeeping
- terminal in webstorm is not unix based but command prompt (cmd.exe) based
- changed github repo
    - update with:

  go get -u github.com/goestoeleven/golangtraining
    - reading docs
        - go
        - go help get

Video #25
Numeral Systems

- understanding numeral systems
- common numeral systems
    - https://docs.google.com/document/d/1GOVIFLkV0TQ49NGgX5_77JJZwJXdLgVb1sGSPk0jsAg/edit?usp=sharing
    - decimal
    - binary
    - hexadecimal
- reason for different numeral systems
- fmt.Printf and formatting verbs
-

Video #26
Binary Numbering System

- how the binary numbering system works
- representing quantities in binary
- hands-on exercises

Video #27
Hexadecimal Numbering System

- how the hexadecimal numbering system works
- representing quantities in hexadecimal
- hands-on exercises
- why we use different numbering systems
    - hex can represent larger quantities in less space
        - hex sometimes has this prefix: 0x
    - computers store binary numbers easily because they have switches that are either on or off

Video #28
Text Encoding

- hexadecimal
    - often called hex

    - sometimes has “0x” prefix indicating it’s hex
- Octal
    - There is a base 8 numbering system called Octal
    - often called Oct
    - Question:
        - how would you represent the number 42 in Oct?
        - Why have Oct?
- ASCII
- UTF

Video #29
Coding Scheme Programs

- format printing
    - a program that prints a quantity in decimal
    - a program that prints a quantity in binary
    - a program that prints a quantity in hex
- format verbs
    - %d
    - %b
    - %x
- escape characters
    - \n
- godoc.org documentation

Video #30
Format Printing

- format verbs
    - %#x
    - %#X
- escape characters
    - \t
- a program that prints a series of numbers in decimal, binary, and hex
    - loop syntax
- a program that prints the first 200 characters of UTF-8
    - loop syntax

Video #31
Section Review

- Housekeeping
    - github
        - git status
        - git add --all
        - git status

        - git commit -m “adds changes to 01 getting started folder”
        - git push
- Review
    - How computers work
        - electricity
        - circuits / switches / transistors
        - coding schemes
            - 2^n
        - binary
        - binary digits = bits
        - measuring bits
            - Bits
            - Bytes
            - KB
            - MB
            - GB
            - TB
        - Machine Language
        - 5 generations of computers
            - vacuum tubes
            - transistors
            - chips
            - cpu’s
            - ???
        - Moore's Law
    - go get
        - updating a package
            - go get -u [package name]
        - reading docs
            - go
            - go help get
    - Numeral systems
        - decimal
        - binary
        - hexadecimal
        - octal
    - Text Encoding
        - ASCII
        - UTF
    - Coding Scheme Programs
        - format printing
          format verbs

            - %d
            - %b
            - %x
        - escape characters
            - \n
            - \t
        - loop
            - printed a series of numbers in decimal, binary, and hex
            - printed the first 200 characters of UTF-8

LANGUAGE FUNDAMENTALS

Video #32
Section Overview

- FYI
    - ardan labs training
- preview
    - packages
    - variables
        - shorthand
        - var → zero value
    - scope
    - blank identifier
    - constants
    - memory addresses (pointers)
    - remainder

Video #33
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 #34
Go Commands

- go commands
    - go run
    - go build
    - go install
    - go clean

Video #35
Variables

- shorthand
    - can only be used inside func
- var
    - zero value
- type format verb
    - %T
- not preferred methods
- declare, assign, initialize

Video #36
scope

- levels of scope
    - universe
    - package
    - file
    - block (curly braces)
- FYI
    - {} - braces, curly braces, curlies, mustaches
    - [] - brackets
    - () - parentheses, parens
- package level scope
    - for variables
    - not for imports
- file level scope
    - imports
- block level scope
- keep your scope tight

Video #37

Scope II

- an example of
    - block level scope
    - order mattering in block level scope
    - variable shadowing

Video #38
Closure

- closure
- variables declared in the outer scope are accessible by statements in the inner scopes which are enclosed by the outer scope

Video #39
Language Spec

- reading the go language specification to reconcile what we have learned with how the language specification talks about it
- some personal information about me, the perfection of imperfection, and our collective humanity

Video #40
Blank Identifier

- you must use everything you put in your code
    - if you declare a variable, you must use it
- the blank identifier
    - _
    - allows you to tell the compiler you aren't using something
- example
    - http.Get
    - throwing away an error

Video #41
Constants

- a simple, unchanging value
- iota's
    - creating constants values for
        - KB
        - MB
        - GB
        - TB

Video #42
Constants II

- a simple, unchanging value
- a parallel type system
    - C / C++ has problems with a lack of strict typing
    - in Go, there is no mixing of numeric types
- 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
        - a “kind”
        - not yet forced to obey the strict rules that prevent combining differently typed values
- 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

Video #43
Words of Encouragement

- You are doing awesome
- You are learning great things which, if not presented correctly, would have been way more challenging

Video #44
Memory Addresses

- memory
    - memory addresses
- Seeing a memory address

Video #45
Pointers

- pointers
    - Using memory addresses in statements

- referencing / dereferencing

#46
Using Pointers
- It's all pass by value

Video #47
Remainder
- remainder
- conditional logic
    - print Odd / Even
- loop
- encouragement
    - building skills - awesome for
        - creating / art
        - job prospects
        - supporting yourself / your loved ones
    - Maynard
        - you got it going on, baby, you got it going on

Video #48
Section Review
- ardan labs training
- HIGH OVERVIEW
    - packages
    - variables
        - shorthand
        - var → zero value
    - scope
    - blank identifier
    - constants
    - memory addresses (pointers)
    - remainder
- 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
- go commands
    - go run
    - go build
    - go install
    - go clean
- variables
    - shorthand
        - can only be used inside func
    - var
        - zero value
    - type format verb
        - %T
    - not preferred methods
    - declare, assign, initialize
- scope
    - levels of scope
        - universe
        - package
        - file
        - block (curly braces)
    - FYI
        - {} - braces, curly braces, curlies, mustaches
        - [] - brackets
        - () - parentheses, parens
    - keep your scope tight
    - **closure**
        - variables declared in the outer scope are accessible by statements in the inner scopes which are enclosed by the outer scope
        - anonymous funcs
        - func expressions
- Blank Identifier
    - tell the compiler you aren't using something
        - you must use everything you put in your code
    - http.Get
        - throwing away an error
- Constants
    - a simple, unchanging value

    - iota's
        - creating constants values for
            - KB
            - MB
            - GB
            - TB
        - bitwise operations
    - a parallel type system
        - 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
            - a "kind"
            - not yet forced to obey the strict rules that prevent combining differently typed values
- Memory Addresses
    - memory
    - memory addresses
    - Seeing a memory address
- Pointers
    - Using memory addresses in statements
    - referencing / dereferencing
    - It's all pass by value
- Remainder
    - remainder
    - conditional logic
        - print Odd / Even
    - loop
- Words of Encouragement
    - You are doing awesome
    - You are learning great things which, if not presented correctly, would have been way more challenging
    - building skills - awesome for
        - creating / art
        - job prospects
        - supporting yourself / your loved ones
    - Maynard
        - you got it going on, baby, you got it going on

    - Some personal information about me, the perfection of imperfection, and our collective humanity

CONTROL FLOW

Video #49
Section Overview

- 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
- Exercises

Video #50
For Loop

- resources
    - https://forum.golangbridge.org/
    - Dave Cheney
- for loop
    - documentation
        - language spec

        - effective go
    - initialization, condition, post

Video #51
Nested Loops

- for loop
    - nested loops

Video #52
Conditions, Break, & Continue

- for loop
    - condition
    - no condition
    - break
    - continue

Video #53
Documentation & Terminology

- documentation
    - golang spec
    - effective go
- terminology
    - lexical elements
    - literal values
    - runes
- rune
    - character
    - an integer value identifying a Unicode code point
    - alias for int32
        - how many bytes in 32 bits? (4 bytes → 4 * 8 = 32)
        - UTF-8 is a 4 byte coding scheme
        - with int32 (4 bytes) we have numbers for each of the code points

Video #54
Rune

- casey neistat
- rune
- printing UTF-8

Video #55
String Type

- using single-quotes

- strings
    - made up of runes

Video #56
Switch Statements
- switch / case / default statements
    - no default fall-through
    - creating fall-through
    - multiple cases
    - cases can be expressions
        - evaluate to true, they run
    - type

Video #57
If Statements
- the not operator
    - !
- initialization statement
- if / else
- if / else if / else
- if / else if / else if / … / else

Video #58
Exercise Solutions
- exercises are presented
- the solutions to the exercises are provided

Video #59
Section Review
- Control Flow
    - sequence
    - loop / iterative
        - nested loops
        - break
        - continue
    - conditionals
        - switch / case / default statements
            - no default 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
- resources
    - https://forum.golangbridge.org/
    - Dave Cheney blog
- documentation
    - language spec
    - effective go
- terminology
    - lexical elements
    - literal values
    - runes
- rune
    - character
    - an integer value identifying a Unicode code point
    - alias for int32
        - UTF-8 is a 4 byte coding scheme
        - with int32 (4 bytes) we have numbers for each of the code points
- strings
    - made up of runes
- Exercise Solutions

FUNCTIONS

Video #60
Section Overview

- functions
    - params
        - multiple “variadic” params
    - args
        - multiple “variadic” args
    - returns
        - multiple returns
        - named returns - yuck!
    - review
        - func expressions
        - closure

        - callbacks
        - recursion
        - defer
        - anonymous self-executing functions
        - pass by value
            - reference types
        - data structures preview
            - maps
            - slices
            - structs
- Exercises

Video #61
Intro To Functions

- func main
- func syntax
    - func, receiver, identifier, params, returns
- purpose of functions
    - abstract code
    - code reusability
- parameters vs. arguments
- declaring a func with multiple params

video #62
Func Returns

- function syntax when declaring a func
- a single return
- a named return
- multiple returns

video #63
Variadic Functions

- a func that accepts an unlimited number of parameters

Video #64
Variadic Arguments

- personal anecdote: head down, ox plowing field; doing the work
- variadic funcs
    - my phrase: variadic parameter
    - my phrase: variadic argument
- different ways of writing the same functionality

video #65

Func Expressions

- assigning a func to a variable

video #66
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

video #67
Callbacks

- 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

video #68
Callback Example

- a cool example of a callback

video #69
Recursion

- a func that calls itself
- factorial example

video #70
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.

video #71
Pass By Value

- in Go, everything is pass by value

video #72
Reference Types

- map
- slice
- with reference types, you do not need to pass an address

video #73

Anonymous Self-Executing Functions

video #74
Bool Expressions

- expressions vs statements
- bool types
    - true, false
- operators
    - not

      !
    - or

      ||
    - and

      &&

video #75 - Part I
Exercises

- Exercises
    - multiple returns
    - func expression
    - variadic func
    - bool & operator

video #75 - Part II
Exercises

- Exercises
    - variadic again
    - project euler

video #75 - Part III
Exercises

- Project Euler
    - https://projecteuler.net/

video #76
Section 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
- closure
    - one scope enclosing another
    - variables declared in the outer scope are accessible in inner scopes
    - closure helps us limit the scope of variables
- callbacks
    - passing a func as an argument
- recursion
    - factorial
- defer
- anonymous self-executing functions
- pass by value
- reference types
    - map
    - slice
    - with reference types, you do not need to pass an address

DATA STRUCTURES - ARRAY

Video #77
Section Overview

- array
    - a numbered sequence of elements of a single type
    - does not change in size
    - https://golang.org/ref/spec#Array_types
- slice
    - a list
    - A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array. A slice

type denotes the set of all slices of arrays of its element type. **The value of an uninitialized slice is nil.**
    - change in size
    - have a length and a capacity
    - multi-dimensional
    - https://golang.org/ref/spec#Slice_types
- map
    - key / value storage
    - a “dictionary”
    - A map is 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. **The value of an uninitialized map is nil.**
    - https://golang.org/ref/spec#Map_types
- struct
    - a data structure
    - a composite type
    - allows us to collect properties together
    - https://golang.org/ref/spec#Struct_types

Video #78
Array

- definition
    - An array is a numbered sequence of elements of a single type.
    - The number of elements is called the length and is never negative.
    - The **length is part of the array's type**; it must evaluate to a non-negative constant representable by a value of type int.
        - The length of an array a can be discovered using the built-in function len.
    - The elements can be addressed by integer indices 0 through len(a)-1.
    - Array types are always one-dimensional but may be composed to form multi-dimensional types.
    - not dynamic
        - does not change in size
- a basic array
    - len
    - index access
    - assigning a value to an index position in an array

video #79
Array Examples

- understanding the difference between index position and the items stored
    - if you’re storing three items in array a, those items will be at index positions 0, 1, 2
        - len(a)-1 is your last index position

            - eg, 3-1 = 2 → 2 is your last index position for your array, a, which has three items
- using break in a loop

DATA STRUCTURES - SLICE

video #80
Slice

- definition
    - A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array.
    - The value of an uninitialized slice is nil.
        - **it is a reference type**
    - Like arrays, slices are indexable and have a length.
    - The length of a slice s can be discovered by the built-in function len;
        - Unlike arrays, slices are dynamic
            - their length may change during execution.
    - The elements can be addressed by integer indices 0 through len(s)-1.
    - A slice, once initialized, is always associated with an underlying array that holds its elements.
        - **it is a reference type**
    - The array underlying a slice may extend past the end of the slice.
        - 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;
        - The capacity of a slice a can be discovered using the built-in function cap(a).
    - make
        - A new, initialized slice value for a given element type T is made using the built-in function make, which takes a slice type and parameters specifying the length and optionally the capacity.
        - A slice created with make always allocates a new, hidden array to which the returned slice value refers.
            - make([]T, length, capacity)
            - make([]int, 50, 100)
                - same as this: new([100]int)[0:50]
    - Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. (multi-dimensional slices)
- a basic slice

video #81
Slice Examples

- length and capacity
    - a great example
- index out of range errors
- appending items to slices
- deleting items from slices

video #82
More Slice Examples

- multidimensional slice
- incrementing a slice

video #83
Creating A Slice

- shorthand
- var
    - sets slice to zero value which is nil
- make

video #84
Incrementing A Slice Item

- incrementing a slice item
- review of slices
    - len, cap, underlying array, append

video #84_02
Section Review

- definition
    - a list of values of a certain Type
- internals
    - reference type
    - pointer, len, cap
    - built on-top of an array
        - another way to say it: “points to an array”
    - The value of an uninitialized slice is nil.
        - **because it is a reference type**
        - A slice, once initialized, is always associated with an underlying array that holds its elements.
        - slices are dynamic (unlike arrays)

            - their length may change during execution.
        - The array underlying a slice may extend past the end of the slice.
            - Capacity is a measure of that extent:
            - The capacity of a slice a can be discovered using the built-in function cap(a).
- make
    - A slice created with make always allocates a new, hidden array to which the returned slice value refers.
        - make([]T, length, capacity)
        - make([]int, 50, 100)
            - same as this: new([100]int)[0:50]
    - Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. (multi-dimensional slices)
- index out of range errors
- appending items to slices
    - access by index if the index is less than the length of the slice less one
        - 0 through len(s)-1.
- deleting items from slices

    mySlice = append(mySlice[:2], mySlice[3:]...)
- incrementing a slice

    mySlice[0]++
- creating a slice
    - shorthand

        student := []string{}
    - var
        - sets slice to zero value which is nil

        var student []string
    - make

        student := make([]string, 35)

DATA STRUCTURES - MAP

video #85
Maps Introduction

- maps
    - key / value storage
    - a “dictionary”
    - 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.
        - **The value of an uninitialized map is nil.**
- reference type

video #86
Map Examples - Part I

- examples
    - creating maps
        - var
        - make
        - shorthand
    - adding entries

video #86
Map Examples - Part II

- examples
    - updating entries

video #86
Map Examples - Part III

- examples
    - deleting entries

video #87
Map Documentation

- documentation
    - golang spec
    - effective go
    - github golang
- comma, ok idiom

Video #88
Map Range Loop

- range loop

Video #89
GitHub Pull

Video #90
Hash Tables

- Macro View of Map Internals In Go
- hash tables explained

Video #91
Hasing Words

- all english words
    - a tour of a program that introduces us to
        - interfaces
        - buffers
        - scanners
    - previews interfaces

video #92
Build A Hash Table

- building a hash table
    - letter buckets
    - remainder buckets
    - hashing words

video #93
building our hash function

- hashing words in more detail
- abstraction / modularizing code
- troubleshooting a conversion error
- bufio.NewScanner
- scanner.Split
- bufio.ScanWords

video #94
finished hash algorithm

- http.Get
- bufio.NewScanner
- scanner.Split
- bufio.ScanWords
- multi-dimensional slice
    - slice of slice of string
- defer
- ++

DATA STRUCTURES - STRUCT

Video #95
Structs Introduction

- introduction to struct
- user-defined types
    - creating values of a certain type
- an example struct
- OOP
    - in other languages
        - classes - blueprint
            - instantiate objects from a class
    - in Go
        - types - blueprint
            - create values of that type

Video #96
OOP in Go

- Documentation
    - golang spec
        - A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). Within a struct, non-blank field names must be unique.
        - A field declared with a type but no explicit field name is an anonymous field, also called an embedded field or an embedding of the type in the struct. An embedded type must be specified as a type name T or as a pointer to a non-interface type name *T, and T itself may not be a pointer type. The unqualified type name acts as the field name.
        - Promoted
- Resources
    - http://www.goinggo.net/2015/09/composition-with-go.html
- OOP
    - Encapsulation
        - state ("fields")
        - behavior ("methods")
        - exported / un-exported
    - Reusability
        - inheritence ("embedded types")
    - Polymorphism
        - interfaces
    - Overriding
        - embedding types as fields in a struct
            - anonymous types
        - outer type and inner type

            - inner type is promoted to the outer type
        - syntactic sugar
        - overriding fields and methods

Video #97
User-Defined Types

- user-defined types
    - Go is statically typed
- language for talking about types and structs

Video #98
Composition

- methods
- embedded types
- struct pointers

Video #99
JSON Marshal

- definition
- www.jsoneditoronline.org
- https://godoc.org/encoding/json
- encoding
    - Marshal / Unmarshal
        - string
    - encode / decode
        - stream
- Marshal
    - basic
    - unexported fields
    - tags

Video #100
JSON Unmarshal

- unmarshal
    - basic
    - tags

Video #101a
JSON Encode

- encoding
    - writer

Video #101b
JSON Decode

- decoding
    - reader

INTERFACES

Video #102
Interfaces

- intro to interfaces
- making money online
    - amazon affiliates
    - always good to make a buck

Video #103
Interfaces

- Examples

Video #104
Interfaces

- The power of interfaces
- hashtable english alphabet revisited
    - ReadAll(r io.Reader)
        - ioutil.ReadAll(res.Body)
    - NewScanner(r io.Reader)
        - bufio.NewScanner(res.Body)

Video #105
Interfaces

- What others say
    - Go In Action
        - Bill Kennedy

Video #106
Interfaces

- What others say
    - The Go Programming Language
        - Donovan & Kernighan

Video #107

Sort package

- Interfaces explored
- Exercise

Video #108
Sort Solution

- sort.Interface
    - Interface interface from sort package
- sort.StringSlice
    - conversion

Video #109
Sort Solution

- sort.Strings()
- sort.Reverse

Video #110
Sort Solution

- sorting []int

Video #111
Empty Interface

- cleaning up your code
    - golint
    - go fmt
    - ./...
- documentation
- examples

Video #112
Method Sets

- receivers
    - value receiver
        - value type
        - pointer type

        - pointer receiver
            - pointer type
- examples
- documentation

Video #113
Conversion vs Assertion

- documentation
- examples

CONCURRENCY

Video #114
Concurrency & WaitGroup

- go routines
- waitgroups
- 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.

Video #115
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.

Video #116
Race Condition

- Race conditions are among the most insidious and elusive programming errors. They typically cause erratic and mysterious failures, often long after the code has been deployed to production. While Go's concurrency mechanisms make it easy to write clean concurrent code, they don't prevent race conditions. Care, diligence, and testing are required. And tools can help.

Video #117
Mutex

- A Mutex is a mutual exclusion lock. Mutexes can be created as part of other structures; the zero value for a Mutex is an unlocked mutex.

Video #118
Atomicity

- Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.
- These functions require great care to be used correctly. Except for special, low-level applications, synchronization is better done with channels or the facilities of the sync package. Share memory by communicating; don't communicate by sharing memory.

Video #119
Review

- reviewing wait groups, mutexes, and atomicity
- resource for learning about channels:
    - https://goo.gl/photos/HQ3xobjN39a3G3ee7

Video #120_01
Channels Introduction

- making a channel
- putting values on a channel
- taking values off of a channel
- buffered and unbuffered channels
- unbuffered 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
    - this is the same with unbuffered channels
    - the value is passed/received synchronously; at the same time
- channels allow us to pass values between goroutines
    - https://goo.gl/photos/HQ3xobjN39a3G3ee7

Video #120_02
Range Clause

- putting values onto a channel
- taking values off of a channel
- closing a channel
- range clause
    - the range clause takes values off a channel
    - when the channel is closed,
        - you can no longer put values on a channel
        - you can still pull any values already on the channel off of the channel
    - when a channel is closed and there are no more values on it
        - the range clause completes
        - flow of the program continues sequentially

Video #120_03
N-to-1

- many funcs in different goroutines writing to the same channel
- troubleshooting a race condition
- using wait groups

Video #120_04
Semaphores

- Getting rid of wait groups
- using semaphores to signal when a goroutine is done

Video #120_05
Semaphores Part 2

- handling N number of goroutines

Video #120_06
1-to-N

- one channel with many funcs receiving from it

Video #120_07
Channels as Arguments & Returns

- you can pass channels between functions
    - you can use channels as arguments
    - you can return channels from functions

Video #120_08
Channel Direction

- channels can have direction
    - bidirection
    - receive only
    - send only

Video #120_09
Incrementor With Channels

- checking for deadlocks
- checking to make sure main doesn't exit before all processing is done

Video #120_10
Deadlock Challenges

- understanding deadlocks
- troubleshooting deadlocks
- writing code that doesn't have deadlocks

Video #120_11
Factorial Challenges

- using goroutines and channels to solve a factorial problem

Video #120_12
Pipeline Pattern

- understanding the pipeline pattern

Video #120_13
Factorial Challenge Redux

- using pipelines for the factorial problem

Video #120_14
Factorial Challenge Redux Solution

- using pipelines for the factorial problem

Video #120_15

Fan Out / Fan In Pattern

- Fan Out: multiple funcs reading from that channel until it's closed
- Fan In: multiple channels writing to the same channel
- https://docs.google.com/presentation/d/1JlYAF8vQLlfuKcnW5gqBw6w4Li0pHk8Kf37UoMTSasY/edit?usp=sharing
- https://blog.golang.org/pipelines

Video #120_16
Fan In Pattern

- multiple funcs reading from that channel until it's closed

Video #120_17
Fan Out / Fan In Example

- an example showing fan out / fan in

Video #120_18_01
Fan out / Fan in Challenge

- Here is some code which someone had put online. They put this code online to illustrate "fan out and fan in". Is this code actually using "fan out and fan in"? Your challenge is to identify whether or not "fan out and fan in" are being used in this code.

Video #120_18_02
Factorial Fan Out / Fan In Challenge

- Here is the solution to the challenge as to whether or not the code found online is using "fan out" and "fan in".

Video #120_19
Factorial Fan Out / Fan In Challenge

- Use fan out and fan in to compute factorial calculations

Video #120_20
Fan Out / Fan In - Solution: Factorial

- The solution to how to calculate factorial with fan out and fan in

Video #120_21
Deadlock Challenge

- This code throws a deadlock error. Can you fix it?

Video #120_22
Deadlock Solution

- Here is the solution to the deadlock challenge

Video #120_23
Incrementor Challenge
- Can you refactor our incrementor code from before to now use channels?

Video #120_24
Incrementor Solution
- Here is how you use channels to implement the incrementor problem.

Video #121
Additional Resources
- Additional resources for learning concurrency in golang

ERROR HANDLING

Video #121_b01_error_intro
- 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

Video #121_b02_golint
- Golint
- Super simple highlighter

Video #121_b03_stdout_stderror
- fmt.Println
- log.Println
- log.SetOutput

Video #121_b04_log-fatal_panic
- Package builtin
- log.Fatalln
- panic

Video #121_b05_errors-new_documentation
- "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
- errors.New
- Type error

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

- Looking at the implementation in the standard library

Video #121_b06_errors-new_assign-to-variable

- Assigning a custom error to a variable
- Prefix your variable with “Err”
    - ErrNorgateMath
    - ErrBufferFull
    - ErrInvalidUnreadByte

Video #121_b07_fmt-errorf

- fmt.Errorf

```
func Errorf(format string, a ...interface{}) error {
    return errors.New(Sprintf(format, a...))
}
```

Video #121_b08_struct-error

- Providing even more context with errors
- Steps:
    - Create a struct
        - Add fields to the struct
        - Add a field of type error to the struct
    - Have the struct implement the error interface
        - Attached this method “Error() string” to the struct
    - Now use your struct as an error type

Video #121_b09_review_resources

- Review
    - errors
        - Go doesn’t favor try / catch
    - golint
    - Handling returned errors
        - fmt.Println("err happened", err)
        - log.Println("err happened", err)
        - log.Fatalln(err)
            - log.SetOutput(w io.Writer)
        - panic(err)

    - errors package
    - builtin package
        - type error
            - Error() string
    - Customizing error message
        - errors.New
        - fmt.Errorf
        - having a struct implement type error
- Resources
    - http://blog.golang.org/error-handling-and-go
    - http://www.goinggo.net/2014/10/error-handling-in-go-part-i.html
    - http://www.goinggo.net/2014/11/error-handling-in-go-part-ii.html
    - https://golang.org/ref/spec#Errors
    - https://golang.org/doc/effective_go.html#errors

FAREWELL

Video #122