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

Course introduction

Welcome

- **Getting resources**
  - next to every video
  - “course resources” in this section
  - https://github.com/GoesToEleven/learn-to-code-go-version-03
    - **WORK IN PROCESS**
- Questions and answers
  - This is THE BEST AND QUICKEST place to get your questions answered
  - golang bridge forum
- Go vs golang
- golang playground
- **Mindset**
  - Be an **adventurer**
    - spirit of exploration
    - don't fear computers - fear ignorance
  - **Practice**

        - practice leads to progress
            - drop by drop …
            - persistently patiently …
            - every day I take consistent action …
            - grit … - Angela Duckworth
        - **Perspective**
            - as you think and act …
            - henry ford, whether you think you can …
            - bill gates, 1 year vs 10 years …
        - **Transformation**
            - education can transform your life
                - my life, the lives of others
            - next video might potentially be the Udemy & Atlassian video
                - or I will try to link to it in this video and all of the videos in this section
- **Imposter syndrome**
    - 50 - 80% of programmers feel this
    - we are always operating on the edge of understanding
        - *we don't have to know the answer, we have to know how to find the answer.*
    - if you don't believe in yourself, move forward regardless
        - the belief will come
- **Band together**
    - we're in this together
        - beginners, intermediate, & advanced
        - be a teacher and a student
            - we can all learn from each other
    - let's help each other and be kind
        - post questions; answer questions
            - it will help you learn

curriculum item # 001-welcome

video: 001-WELCOME-take-02.mp4

Course resources

curriculum item # 002a no video

The Power of Education to Transform Lives

Additional course resources

curriculum item # 002b no video

The Power of Education to Transform Lives

The Power of Education to Transform Lives

curriculum item # 003 no video

Getting going with Go

Why Go?

- built by 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
- timeline
    - 2005 - first dual core processors
    - 2006 - Go development started
    - 2009 - open sourced (november)
    - 2012 - version 1 (march) - go blog link
- Why Go
    - **efficient compilation**
        - Go creates compiled programs
            - there is a garbage collector (GC)
            - there is no virtual machine
        - fast compilation → fast development
        - compiles to a **static** executable
            - **static linking** - compiles libraries into your program.
            - **dynamic linking** loads libraries into memory when program runs
        - cross compiles to different OS's
    - **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
    - robust and mature standard library
    - third party packages
    - stack overflow developer survey
- Guiding principles of design
    - expressive, comprehensible, sophisticated
    - clean, clear, easy to read
- notables
    - friendly and welcoming community!
    - open source
        - go is written in go
    - backward compatibility promise
        - from version 1 onward
    - built in HTTP server that is production ready
    - static types (not dynamic)
        - a VALUE of a certain TYPE
    - garbage collector - fast!
        - memory is managed by go, and not by you
    - tooling
        - the go tool
        - modules & dependency management
    - great documentation
        - written by the masters
            - **golang documentation**
            - **golang specification**
            - **effective go**
            - **golang user manual**
            - **standard library**
            - **golang blog**
            - **golang modules**
                - **https://go.dev/blog/using-go-modules**
                - **https://go.dev/ref/mod**
            - **go help at command line**
    - mechanical sympathy
- The language is “GO”
    - as in, go fast
    - as in, **GO**OGLE
    - but when google about Go, use golang
- Super cute mascot
    - The **Go**pher
        - https://github.com/ashleymcnamara/gophers
        - https://github.com/egonelbre/gophers

Go, also known as Golang, is a modern programming language developed by Google in 2007. It has quickly become popular among developers for its simplicity, efficiency, and powerful features.

One of the key features of Go is its simplicity. The language has a clean syntax that is easy to read and write, making it an ideal choice for beginners who are just starting to learn programming. Despite its simplicity, Go is a powerful language that can handle complex tasks and scale well, making it an excellent choice for building large-scale software projects.

Another advantage of Go is its performance. Go is a compiled language, which means that it can produce highly optimized machine code that runs very quickly. This makes Go an ideal choice for building high-performance applications, such as network servers, web applications, and real-time systems.

Go also comes with a built-in concurrency model that makes it easy to write highly concurrent and parallel programs. This makes it an ideal choice for building applications that need to handle a large number of concurrent requests or perform multiple tasks simultaneously.

Go is also an open-source language, which means that it is constantly evolving and improving with contributions from a large and active community of developers. This ensures that Go will remain relevant and up-to-date with the latest developments in technology.

Overall, Go is a great language to learn today because it is simple, powerful, fast, concurrent, and constantly evolving. Whether you are a beginner or an experienced programmer, Go is a language that can help you build great software projects efficiently and effectively.

curriculum item # 004-why-go

video: 004-why-go-EDIT-02.mp4

Documentation & example code

Documentation

- **golang documentation**
    - **golang user manual**
- **golang specification**
- **effective go**
- **standard library**
- **golang blog**
- **golang modules**
    - **How to Write Go Code**
    - **Tutorial: Create a Go module**
    - **https://go.dev/blog/using-go-modules**
    - **https://go.dev/ref/mod**
    - **also great**

        - **digital ocean tutorial go modules**
            - **https://www.digitalocean.com/community/tutorials/how-to-use-go-modules**
        - **practical go lessons chapter 17 - go modules**
            - **https://www.practical-go-lessons.com/chap-17-go-modules**
- **go help at command line**
- **go proverbs**
- **a tour of go**

Example code

- golang by example
    - https://gobyexample.com/

curriculum item # 005--docs–and-example-code

video: 005-docs–and-example-code-EDIT-02.mp4

Creating our first go program - hello gophers 😃!

- in this video
    - comments
    - package main
    - func main
    - fmt package
    - semicolons are inserted by compiler

QUICK SETUP GUIDE

- (for those who are experienced)
- (for everyone else: we’ll learn how to run code on our machines soon)
    - install go
    - install git-scm
    - install vs code
        - installing go plugin
            - built and maintained by google
        - updating tooling
            - command palette / go install update tools
    - go modules
        - go mod init <AnyNameYouWant>
        - blog
            - https://go.dev/blog/all (ctrl+f “module”)
        - tutorial
            - https://go.dev/doc/tutorial/create-module
        - managing dependencies
            - https://go.dev/doc/modules/managing-dependencies#naming_module

https://go.dev/play/p/dRzQghWc-gA

curriculum item # 006-package-main-func-main

video: 006-package-main-func-main-02.mp4

Exploring format printing and documentation

- 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{}**"
        - "any" is of type interface
    - **parameter** "...any"
        - means give me as many **arguments** of any type as you'd like
- throwing away returns
    - use **the "_" underscore character** to throw away returns
    - this is called "the blank identifier"
- 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://go.dev/play/p/wjiO9797zXC

curriculum item # 007-format-printing

video: 007-format-printing-02.mp4

How computers work - core principles

- computers run on electricity
- electricity has two discrete states: ON and OFF
- we can create **coding schemes** for "on" or "off" states
    - examples: morse code, halloween

curriculum item # 008-how-computers-work

video: 008-how-computers-work.mp4 **was** 012-how-computers-work.mp4

ASCII, Unicode, & UTF-8 - understanding text

https://go.dev/play/p/PnzhfYhGWCC

curriculum item # 009-ascii-unicode-utf8

video: 009-ascii-uni.mp4 **was** 013-ascii-uni.mp4

String literals and documentation

https://go.dev/play/p/wge8tWLEQNM

curriculum item # 010-string-literals

video: 019-string-literals.mp4 **was** 014-string-literals.mp4

Hands-on exercises

Hands-on exercise #1

- print out a some text with emojis
- print out a raw string literal

https://go.dev/play/p/niidn0N9RHL

curriculum item # 011-exercise-01

video: 011-exercise-01.mp4 **was** 015-exercise-01.mp4

The fundamentals of Go

Variables, zero values, blank identifier

Declaration, assignment, initialization

- general guideline
  - var for zero value
  - short declaration operator
  - var when specificity is required
- multiple initializations

VALUE and TYPE

- go is statically typed, not dynamic
- a **VARIABLE** holds a **VALUE** of a specific **TYPE**

you can't have unused variables in your code

- this is "code pollution"
- the compiler doesn't allow it

zero values

- false boolean
- 0 int
- 0.0 float
- "" string
- nil
  - pointers
  - functions
  - interfaces
  - slices
  - channels
  - maps

https://go.dev/play/p/C5k8w07Fh7Y

curriculum item # 012-variables-zero-val-blank-identifier

video: 012-variables-zero-val-blank-identifier.mp4

Using printf for decimal & hexadecimal values

documentation - know the truth, written by genius

https://go.dev/play/p/uvggBjVF47O

curriculum item # 013-printf-decimal-hex

video: 013-printf-dec.mp4 **was** 018-printf-dec.mp4

Numeral systems: decimal, binary, & hexadecimal

decimal

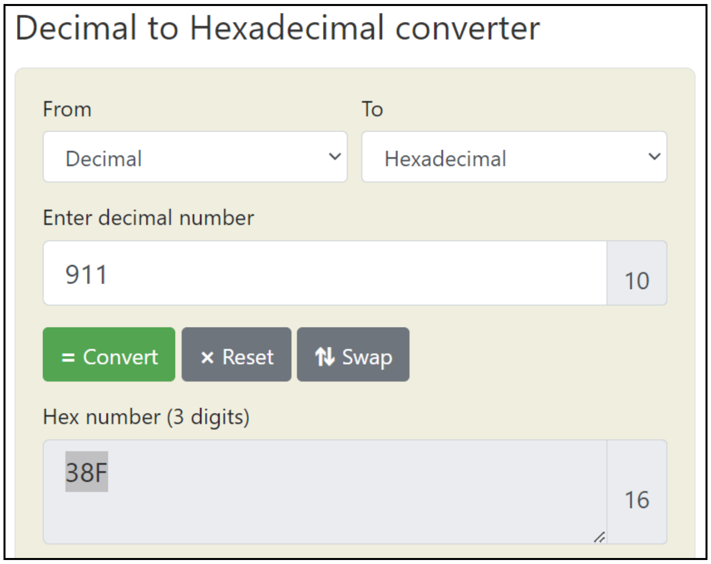
- base 10
- 0,1,2,3,4,5,6,7,8,9

binary

- base 2
- 0,1

hexadecimal

- base 16
- 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- license plate on a Porsche 911 → 38F



Decimal to Hexadecimal converter

From: Decimal

To: Hexadecimal

Enter decimal number: 911 (10)

Hex number (3 digits): 38F (16)

understanding numeral systems

curriculum item # 014-numeral-systems

video: 014-numeral-systems.mp4 **was** 019-numeral-systems.mp4

Values, types, conversion, scope & housekeeping

documentation - know the truth, written by genius

- conversion
    - A conversion changes the type of an **expression** to the type specified by the conversion.

    - A conversion may appear literally in the source, or it may be **implied** by the context in which an expression appears.
    - An **explicit** conversion is an expression of the form T(x) where T is a type and x is an expression that can be converted to type T.
- expressions
    - An expression specifies the computation of a value by applying operators and functions to operands.
- constants (effective go)
    - Constants in Go are just that—constant. They are created at compile time, even when defined as locals in functions, and can only be numbers, characters (runes), strings or booleans. Because of the compile-time restriction, the expressions that define them must be constant expressions, evaluatable by the compiler. For instance, 1<<3 is a constant expression, while math.Sin(math.Pi/4) is not because the function call to math.Sin needs to happen at run time.
- constants (language specification)
    - Constants may be typed or untyped.
    - A constant may be given a type explicitly by a constant declaration or conversion, or implicitly when used in a variable declaration or an assignment statement or as an operand in an expression. It is an error if the constant value cannot be represented as a value of the respective type.
    - An untyped constant has a default type which is the type to which the constant is implicitly converted in contexts where a typed value is required, for instance, in a short variable declaration such as i := 0 where there is no explicit type. The default type of an untyped constant is bool, rune, int, float64, complex128 or string respectively, depending on whether it is a boolean, rune, integer, floating-point, complex, or string constant.
- constants (go blog - ROB PIKE!!)
    - Go is a statically typed language that does not permit operations that mix numeric types. You can’t add a float64 to an int, or even an int32 to an int. Yet it is legal to write 1e6*time.Second or math.Exp(1) or even 1<<(' '+2.0). In Go, constants, unlike variables, behave pretty much like regular numbers. This post explains why that is and what it means.
    - In the early days of thinking about Go, we talked about a number of problems caused by the way C and its descendants let you mix and match numeric types. Many mysterious bugs, crashes, and portability problems are caused by expressions that combine integers of different sizes and “signedness”.
- declarations and scope
    - The scope of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, label, or package.
    - Go is lexically scoped using blocks:
    - The scope of a predeclared identifier is the **universe block.**
    - The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the **package block.**
    - The scope of the package name of an imported package is the **file block** of the file containing the import declaration.
    - The scope of an identifier denoting a method receiver, function parameter, or result variable is the **function body.**

    - The scope of an identifier denoting a type parameter of a function or declared by a method receiver begins after the name of the function and ends at the end of the **function body.**
    - The scope of an identifier denoting a type parameter of a type begins after the name of the type and ends at the end of the TypeSpec.
    - The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block.
    - The scope of a type identifier declared inside a function begins at the identifier in the TypeSpec and ends at the end of the innermost containing block.
    - if statements

```
if x := f(); x < y {
        return x
} else if x > z {
        return z
} else {
        return y
}
```

- unused imports, variables, & the blank identifier (effective go)
    - It is an error to import a package or to declare a variable without using it. Unused imports bloat the program and slow compilation, while a variable that is initialized but not used is at least a wasted computation and perhaps indicative of a larger bug. When a program is under active development, however, unused imports and variables often arise and it can be annoying to delete them just to have the compilation proceed, only to have them be needed again later. The blank identifier provides a workaround.

language spec (ctrl+f “cast” & ctrl+f “conversion”)

https://go.dev/play/p/zNKmoCbPvBm

curriculum item # 015-val-type-conversion-short-dec

video: 015-val-type-conversion-short-dec.mp4 **was** 020-val-type-conversion-short-dec.mp4

Built-in types, aggregate types, and composition

We DECLARE a VARIABLE of a certain TYPE

- it can only hold VALUES of that TYPE
- go is statically typed

basic type / built-in type / primitive type

- 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.
- data type for which the programming language provides built-in support.
- In most programming languages, all basic data types are built-in.
- https://en.wikipedia.org/wiki/Primitive_data_type
- standard library / builtin - https://pkg.go.dev/builtin

aggregate type

- aggregates many values together
    - example: array, slice, struct
- **composite / compound / structure / struct type**
    - include values of different types
    - example: struct
- The act of constructing a STRUCT which is a composite type is known as **composition**
    - many different types into one structure which composes all of those different types together into that one data structure - it's a data STRUCTURE which holds VALUES of many different TYPES
    - combining multiple smaller types into a larger type.
    - embedding types and inner-type promotion
        - (inheriting fields and methods of embedded type)

Golang specification - types

- Boolean
- Numeric
- String
- Array
- Slice
- Struct
- Pointer
- Function
- Interface
- Map
- Channel

Constants

- **typed & untyped constants**
    - avoids subtle bugs
    - constants (go blog - rob pike)
        - Go is a statically typed language that does not permit operations that mix numeric types. You can’t add a float64 to an int, or even an int32 to an int. Yet it is legal to write 1e6*time.Second or math.Exp(1) or even 1<<(' '+2.0). In Go, constants, unlike variables, behave pretty much like regular numbers. This post explains why that is and what it means.
        - In the early days of thinking about Go, we talked about a number of problems caused by the way C and its descendants let you mix and match numeric types. Many mysterious bugs, crashes, and portability problems are caused by expressions that combine integers of different sizes and “signedness”.
    - https://go.dev/blog/constants (ctrl+f “type”)

curriculum item # 016-val-type-documentation

video: 016-val-type-documentation.mp4 **was** 021-val-type-documentation.mp4

Hands-on exercises

Hands-on exercise #2 - go tour step 1 - 3

Do the "tour of go" through step 3

- https://go.dev/tour/list
    - begin - welcome https://go.dev/tour/welcome/1
        - https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html
    - begin - Packages, variables, and functions https://go.dev/tour/basics/1
    - end:

Notes

good notes!
Inside a function, the := short assignment statement can be used in place of a var declaration with implicit type. Outside a function, every statement begins with a keyword (var, func, and so on) and so the := construct is not available. https://go.dev/tour/basics/10
The int, uint, and uintptr types are usually 32 bits wide on 32-bit systems and 64 bits wide on 64-bit systems. When you need an integer value you should use int unless you have a specific reason to use a sized or unsigned integer type. https://go.dev/tour/basics/11
Constants are declared like variables, but with the const keyword. Constants can be character, string, boolean, or numeric values. Constants cannot be declared using the := syntax. https://go.dev/tour/basics/15
An untyped constant takes the type needed by its context. https://go.dev/tour/basics/16

bitwise ops - https://go.dev/play/p/JXA-lkTdU81

curriculum item # 017-hands-on-exercise-02

video: 017-hoe-02-step-1-3.mp4 **was** 022-hoe-02-step-1-3.mp4

Hands-on exercise #3 - go tour step 4 - 7

Do the "tour of go" through step 4 - 7

curriculum item # 018-hands-on-exercise-03

video: 018-hoe-03-step-4-7.mp4 **was** 023-hoe-03-step-4-7.mp4

Hands-on exercise #4 - go tour step 8 - 10

Do the "tour of go" through step 8 - 10

curriculum item # 019-hands-on-exercise-04

video: 019-hoe-04-step-8-10.mp4 **was** 024-hoe-04-step-8-10.mp4

Hands-on exercise #5 - go tour step 11 - 13

Do the "tour of go" through step 11 - 13

curriculum item # 020-hands-on-exercise-05

video: 020-hoe-05-step-11-13.mp4 **was** 025-hoe-05-step-11-13.mp4

Hands-on exercise #6 - go tour step 14 - 15

Do the "tour of go" through step 14 - 15

curriculum item # 021-hands-on-exercise-06

video: 021-hoe-06-step-14-15.mp4 **was** 026-hoe-06-step-14-15.mp4

Hands-on exercise #7 - go tour step 16 - 17 - bitwise ops & bit shifting

Do the "tour of go" through step 16 - 17

curriculum item # 022-hands-on-exercise-07

video: 022-hoe-07-step-16-17-bitwise.mp4 **was** 027-hoe-07-step-16-17-bitwise.mp4

Hands-on exercise #8 (was #03) - iota

We learned about bitwise operations in the last video. Now let's learn about "iota" and use it in a program

- to learn about iota
  - golang spec
  - effective go
- modify this program to use iota
  - https://go.dev/play/p/jZUmqIhqaIC
  - **(note how iota only needs to be mentioned at the top of a block of code)**

https://go.dev/play/p/IGky4zYtPH9

curriculum item # 023-hands-on-exercise-08

video: 023-hands-on-exercise-08.mp4 was 023-hands-on-exercise-03.mp4

Hands-on exercise #9 (was #04) - measuring bits with bitwise operations

create a program that uses iota to calculate the size of **each measurement of bytes**

○ KB	1024 bytes
○ MB	1024 KB

    - GB 1024 MB
    - TB 1024 GB
    - PB 1024 TB
    - EB 1024 EB
- show the sizes of each in DECIMAL and BINARY using fmt.Printf
- hint: use int and not float64 as shown in effective go (we aren't going up to the larger values of ZB and YB so we don't need to capacity of float64)
- hint: the starting code for this is already in effective go

https://go.dev/play/p/RoPv2HjoALg

curriculum item # 024-hands-on-exercise-09

video: 024-hands-on-exercise-09.mp4 was 024-hands-on-exercise-04.mp4

Hands-on exercise #10 (was #05) - zero value, :=, type specificity, blank identifier

write a program that uses the following:

- var for zero value
- short declaration operator
- multiple initializations
- var when specificity is required
- blank identifier

print items as necessary to make the program interesting

https://go.dev/play/p/fr9O2i6Hvvg

curriculum item # 025-hands-on-exercise-10

video: 025-hands-on-exercise-10.mp4 was 025-hands-on-exercise-05.mp4

Hands-on exercise #11 (was #06) - printf verbs to show values and types

write a program that uses the following:

- for a VARIABLE storing a VALUE of TYPE
    - string
    - int
    - float64
- use print verbs to show
    - value
    - type

https://go.dev/play/p/7iZ74Hw_LLa

curriculum item # 026-hands-on-exercise-11

video: 026-hands-on-exercise-11.mp4 was 026-hands-on-exercise-06.mp4

Hands-on exercise #12 (was #07) - printf binary, decimal, & hexadecimal

write a program that uses print verbs to show the following numbers

- 747
- 911
- 90210

as

- decimal
- binary
- hexadecimal

https://go.dev/play/p/7cFkjvbpKuY

curriculum item # 027-hands-on-exercise-12

video: 027-hands-on-exercise-12.mp4 was 027-hands-on-exercise-07.mp4

Hands-on exercise #13 (was #08) - signed and unsigned int

write a program that declares two variables

- one variable to store a VALUE of TYPE **int8**
    - assign to it the largest number possible, then print it
- one variable to store a VALUE of TYPE **uint8**
    - assign to it the largest number possible, then print it

https://go.dev/play/p/-q6V8dbGkgz

curriculum item # 028-hands-on-exercise-13

video: 028-hands-on-exercise-13.mp4 was 028-hands-on-exercise-08.mp4

Programming fundamentals for beginners

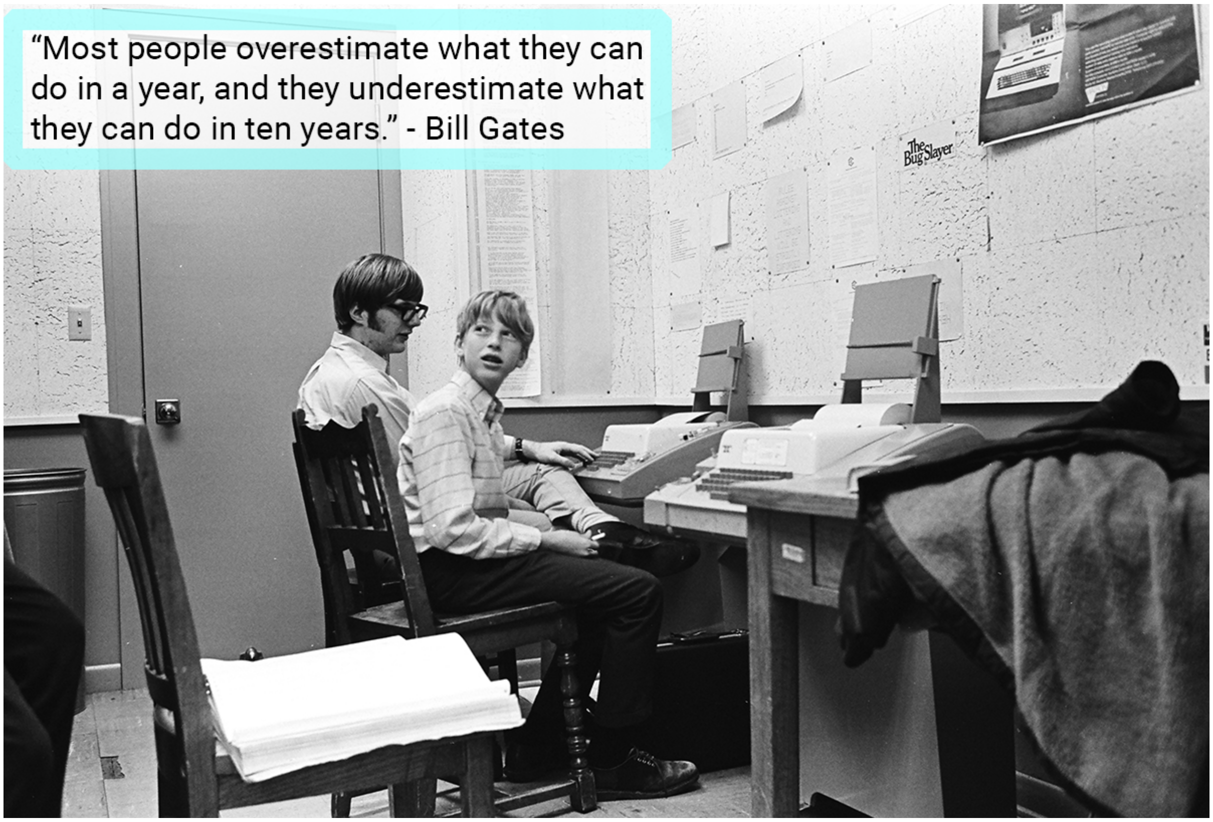
Introduction

- good stuff coming!
    - code preview of scope
- We are riding a rocket ship of a mystery!
    - Earth is traveling through space at 67,000 mph
    - Earth is spinning at 1,000 mph
    - Earth is circling a ball of fire in the sky (the sun)
        - the sun is 1.3 million times larger than earth
    - wow!

curriculum item # 029-intro-programming-fundamentals

video: 029-intro-programming-fundamentals.mp4

Terminology



“Most people overestimate what they can do in a year, and they underestimate what they can do in ten years.” - Bill Gates

- **declare**
  - declare a variable to hold a VALUE of a certain TYPE

```
var x int
```

- **assign**
  - assign a value to a variable

```
var x int = 42
```

- **initialize**
  - declare & assign
  - assign an ***initial*** value to a variable

```
var x int = 42
```

```
var y int
```

```
y = 43
```

- **identifier**
    - the name of a variable
    - examples: x, fName
- **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
        - https://go.dev/ref/spec#Keywords
- **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**
    - 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.

```
x := 2+7
```

- **expression**
    - a value, or operations (operands and operators) that express a value
    - 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+7 is an expression which evaluates to 9. For example, 42 is an expression as it's a value.

```
x := 2+7
y := 42
```

- **parens**

  # ( )

- **curly braces “curlies”**

  { }
- **brackets**

  [ ]
- **scope**
    - where a variable exists and is accessible
    - best practice: keep scope as “narrow” as possible

curriculum item # 030-terminology

Understanding scope

- exploring scope
- https://go.dev/ref/spec#Declarations_and_scope

https://go.dev/play/p/0sU5YFvI1bR
curriculum item # 031-scope

Working at the terminal

- terminology
    - GUI = graphical user interface
    - CLI = command line interface
            - **unix / linux / mac**
                - **shell / bash / terminal**
            - **windows**
                - **command prompt / windows command / cmd / dos prompt**

curriculum item # 032-the-terminal

Using bash on Windows

- https://git-scm.com/
    - start / bash

curriculum item # 033-bash-on-windows

Terminal commands - part 1

an introduction and overview
curriculum item # 034-terminal-commands-pt-1

Terminal commands - part 2

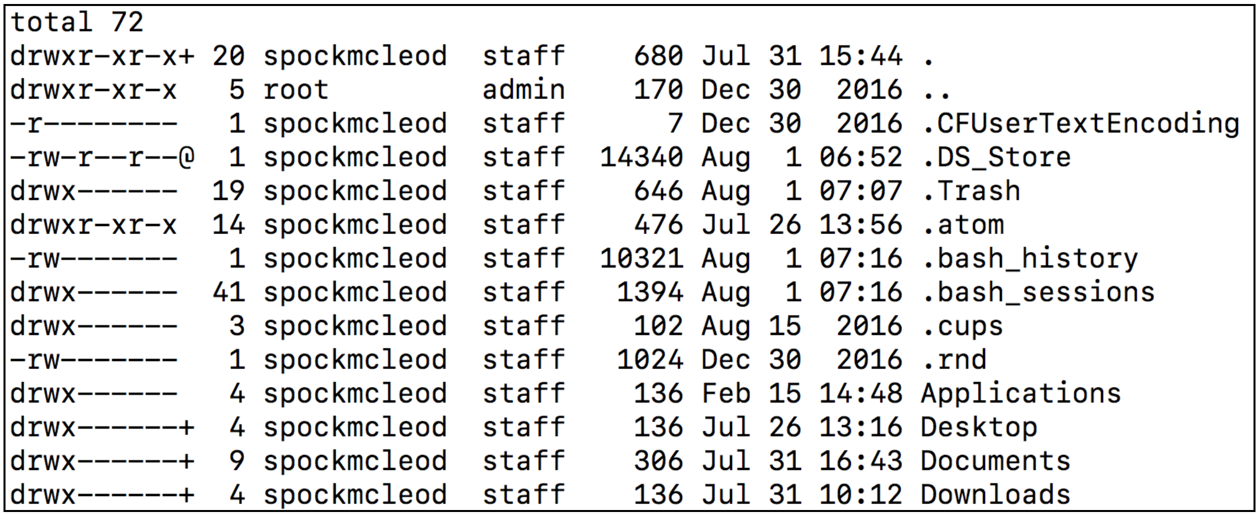
- shell / bash commands

- **pwd**
- **ls**
- **ls -la**
- **ls -lh**
    - 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**
- **chmod**
    - chmod options permissions filename
    - chmod 777 temp.txt
- **env**
    - **go version**
    - **go env**
- **rm <file or folder name>**
    - **rm -rf <file or folder name>**
- **see info on a file**
    - **file <filename>**

    - **LOVE THESE**
        - ulimit -a
        - ps aux | wc -l
            - all of the processes running on your machine
        - lscpu
            - only on linux/mac
            - info about your hardware
        -
    - all commands

curriculum item # 035-terminal-commands-pt-2

Github and ssh authentication

This changes over time, so google for current best practice: “ssh authentication github”
curriculum item # 036-github-ssh

Setting up a github repo

- creating a repo
    - create a repo on github.com
    - create a folder with the same name on your computer
    - follow the instructions from github, from when you created your repo to connect the repo ON YOUR COMPUTER with the repo on GITHUB
- git commands
    - git status
    - git add --all
    - git commit -m “some message”
    - git push

curriculum item # 037-github-setup

Checksums

A checksum in programming is a technique used to ensure the integrity of data that is being transmitted or stored. It is a mathematical algorithm that calculates a fixed-size value based on the content of a file or data stream. This value is then compared to the expected value to ensure that the data has not been altered or corrupted during transmission or storage.

Checksums are commonly used in network protocols such as TCP/IP, as well as in file transfer protocols such as FTP and HTTP. They are also used in computer storage systems, such as hard drives and solid-state drives, to detect errors in data storage.

The most commonly used checksum algorithms include MD5, SHA-1, SHA-256, and CRC (Cyclic Redundancy Check). Each algorithm produces a unique checksum value based on the input data, which can be used to verify the integrity of the data.

In summary, checksums are an important tool in programming to ensure the accuracy and security of data transmission and storage.
curriculum item # 038-checksums

Your development environment

Getting up and running

- installing go
- installing git-scm
    - go uses git when fetching external packages
- installing vs code
    - installing go plugin
        - built and maintained by google
    - updating tooling
        - command palette / go install update tools
    - open a folder

curriculum item # 039-up-and-running

Running go programs on your machine

- create folder
- go mod init <some name>
- create go file
- package main
    - folders are packages
    - every file in a folder/package must have the same package name
- func main
    - dot notation
        - package.function
    - CAPITALIZATION
        - Visible outside package
        - notVisible outside package
    - semi-colons;
        - added by compiler
- running code
    - vs code / run
        - view / debug console
    - command line
        - go run main.go
            - go run ./…

            - builds executable and runs it
        - go build
            - builds executable
- cross build/compile
    - see environment variables
        - go env GOARCH GOOS
        - https://play.golang.org/p/1vp5DImIMM
    - run one of these at the command line to build to a certain OS:
        - GOOS=darwin go build
        - GOOS=linux go build
        - GOOS=windows go build
- go help
    - go help env
    - go help environment
- go env
    - gopath
        - $GOPATH
        - bin folder
- go version

curriculum item # 040-running-go-programs

Go install puts binary in $GOPATH/bin

- go install drops a binary into your $GOPATH bin folder

The go install command in the Go programming language is used to compile and install a package or a set of packages. When you execute go install in a directory containing Go code, it compiles the Go package and installs the resulting binary executable file in the bin directory of your Go workspace.

Specifically, go install does the following:

It compiles the Go package(s) in the current directory and its subdirectories, if any.
It links the compiled object files into a single binary executable file.
It installs the binary executable file in the bin directory of your Go workspace, which is usually located at $GOPATH/bin.
After executing go install, you can run the resulting binary executable file directly from the command line, assuming the $GOPATH/bin directory is included in your system's PATH environment variable.
curriculum item # 041-go-install

Go mod and dependency management

Introduction to go modules & dependency management

Dependency management is the process of identifying, organizing, and resolving the ***external*** code libraries and packages that a software application or project depends upon. Most software applications depend on a large number of external code libraries or packages, which are typically maintained by *third-party developers*. **Dependency management** is important because it ensures that all required libraries and packages are available and compatible with each other, and that any ***updates or changes to those dependencies are managed and controlled to prevent issues that could arise due to conflicting versions or changes.*** There are various ***tools and techniques*** for managing dependencies in programming, including package managers such as **npm** for Node.js, **pip** for Python, and **Maven** for Java. These tools allow developers to easily install, update, and remove external dependencies for their projects, and also provide features like version control, dependency resolution, and automatic updates.

- structured programming
    - spaghetti code
    - modular code
- dependencies
    - your code depends upon other code
        - third-party packages
- go get
    - using the "go get" tool to
        - get third-party packages
        - update third-party packages
        - use a certain version / commit hash of a third-party package
- hands-on
    - code base 1
        - code base 2
            - code base 3

    Using the code base for our course
- modules & packages
    - in go we create modules
        - modules have packages
    - we often push these modules to a code repository
- name spacing
    - domain/username/repo
    - github.com/GoesToEleven/dog
- versions
    - semantic versioning
    - tagging our git commits with versions

    - being able to "go get" a specific
        - commit
        - version
- audience specifics
    - beginners & experienced in this course
    - primary takeaway
        - the concepts
        - go mod init <some-module-name>
- spirit of adventure & exploration
    - tooling
        - @latest doesn't work once
            - we'll see how to force it with @<commit hash>
    - my proficiencies
        - I don't use all of this in my day-to-day, teaching intro programming, so I'm not 100% fluid with the commands, ***but I understand it,*** and we'll get it working!
            - blacksmith analogy
            - we write the code with errors before we write the code without errors
- dependency management problem
    - Russ Cox
- Why now?
    - we've gotten a taste of coding in go
    - now "go mod init <module-name>" is essential to coding on your machine
        - let's understand it
        - *this is not explained well anywhere else*
- Best other resource
    - **digital ocean tutorial go modules**
        - **https://www.digitalocean.com/community/tutorials/how-to-use-go-modules**
- old GO PATH

curriculum item # 042-intro-go-mod-dep-mgmt

Modular code, dependency mgmt, go get - overview

- modular code
    - spaghetti code
        - **Spaghetti code** refers to a programming code that is complex, tangled, and difficult to understand and maintain. The term "spaghetti code" is often used to describe code that has been written in a haphazard and unstructured manner, making it difficult for other programmers to understand and modify the code. Spaghetti code typically arises when the programmer does not follow a structured approach to software development and fails to organize the code into modular, reusable components. This can lead to code that is difficult to read, debug, and maintain over time, which can result in software that is unreliable, inefficient, and prone to errors. The term "spaghetti code" is derived from the visual appearance of the code, which resembles a tangled mess of lines that are difficult to trace

and follow, similar to a plate of spaghetti. **To avoid spaghetti code,** it is important for programmers to follow best practices for software development, such as using a **structured approach**, creating **modular components**, and **documenting** their code thoroughly.

- modular code / structured programming
    - A **structured approach** to software development is a methodical and organized approach to writing software code. It involves breaking down a large software project into smaller, manageable tasks, and designing and implementing each task in a logical and coherent manner. The goal of a structured approach is to make the code more readable, maintainable, and scalable, which helps to reduce errors and improve software quality. **Modular code** is an essential aspect of a structured approach to software development. Modular code refers to a programming technique where ***a large program is divided into smaller, independent modules or components. Each module performs a specific task, and these modules are designed to work together to achieve the overall functionality of the program.*** By breaking down a large program into smaller modules, it becomes easier to manage, test, and maintain the code. Each module can be developed and tested independently, making it easier to debug and modify the code. Additionally, **modular code** can be reused in other projects, saving time and effort in future software development. A **structured approach** to software development involves using best practices such as defining requirements, creating a detailed design, and breaking down the project into smaller, manageable tasks. This approach is intended to help developers produce more reliable, efficient, and scalable code, which is easier to maintain and modify over time. By using modular code, developers can create more structured and maintainable software applications, which are essential for large and complex software projects.

- dependencies
    - code you code depends upon
        - Direct
            - a dependency which your code directly imports.
        - Indirect
            - a dependency not directly used in your code
            - dependency used by your direct dependencies
    - https://go.dev/ref/mod#glos-direct-dependency
- go mod
    - configures our go workspace
        - examples
            - go mod init mymodule
            - go mod init github.com/GoesToEleven/animalPackage
    - helps us manage dependencies
        - dependencies = other code our project depends on
- dependencies & security considerations
    - Russ Cox - Our Software Dependency Problem (01/23/2019)
- go get
    - allows to go get a third-party package dependency to use in our code

https://github.com/astaxie/build-web-application-with-golang
https://www.golang-book.com/books/intro
curriculum item # 043-mod-code-depend-overview

Go modules in action: go mod init & go mod tidy

go mod tidy is a command used in Go programming language to manage dependencies in a project. When you create a Go module, you may add dependencies to your project using third-party packages. These packages can be installed automatically when you run your code or explicitly by using the go get command.

The go mod tidy command checks the go.mod file in your project, and it ensures that it includes all the necessary dependencies for the project to build and run correctly. It also removes any unused dependencies that are no longer required by the project.

In more detail, when you run the go mod tidy command, it performs the following steps:

- It reads the go.mod file and identifies all the dependencies listed in it.
- It checks if all the dependencies listed in the go.mod file are required for the project. If any of the dependencies are not required, it removes them from the go.mod file.
- It adds any missing dependencies to the go.mod file that are required by the project.
- It verifies that all the dependencies have the correct versions specified in the go.mod file.
- It updates the go.sum file to include the hashes of the downloaded dependencies.

Overall, the go mod tidy command helps to ensure that your Go project has the correct dependencies specified, and it removes any unused dependencies that may slow down your project or cause compatibility issues.
curriculum item # 044-go-mod-in-action

Looking at the documentation for go mod tidy

go mod tidy is a command used in Go programming language to manage dependencies in a project. When you create a Go module, you may add dependencies to your project using third-party packages. These packages can be installed automatically when you run your code or explicitly by using the go get command.

The go mod tidy command checks the go.mod file in your project, and it ensures that it includes all the necessary dependencies for the project to build and run correctly. It also removes any unused dependencies that are no longer required by the project.

In more detail, when you run the go mod tidy command, it performs the following steps:

- It reads the go.mod file and identifies all the dependencies listed in it.
- It checks if all the dependencies listed in the go.mod file are required for the project. If any of the dependencies are not required, it removes them from the go.mod file.
- It adds any missing dependencies to the go.mod file that are required by the project.
- It verifies that all the dependencies have the correct versions specified in the go.mod file.
- It updates the go.sum file to include the hashes of the downloaded dependencies.

Overall, the go mod tidy command helps to ensure that your Go project has the correct dependencies specified, and it removes any unused dependencies that may slow down your project or cause compatibility issues.
curriculum item # 045-go-mod-tidy-documentation

Modular code, dependency mgmt, go get - #1

- naming packages
    - https://go.dev/blog/package-names
- package puppy
    - https://github.com/GoesToEleven/puppy
    - add functions
    - push
- learn-to-code-go-version-03
    - https://github.com/GoesToEleven/learn-to-code-go-version-03
    - inspect
        - go.mod
        - go.sum
    - go get github.com/GoesToEleven/puppy
        - go get github.com/GoesToEleven/puppy@latest
    - inspect
        - go.mod
        - go.sum
    - use functions from puppy

learn-to-code-go-version-03 ← puppy ← dog

In Go, modular code refers to breaking up a program into smaller, reusable modules that can be easily managed and maintained. This approach helps developers write more organized and maintainable code by focusing on building independent modules that perform a specific task or have a specific functionality.

Dependency management is the process of handling external libraries and packages that a Go program relies on. When developing a Go program, developers often use third-party libraries to speed up development or add specific features. Dependency management refers to managing these external libraries to ensure that the program runs as expected, and updates to the libraries don't break the program's functionality.

The Go programming language has a built-in package manager called "go modules." The module system provides a way to manage dependencies at the module level, allowing developers to specify the exact version of a dependency that their program requires. The module system also provides a way to easily download and manage dependencies for a Go project, making it easy to build modular and maintainable code in Go.

https://go.dev/play/p/pRDSWJzN5Xr
curriculum item # 046-mod-code-depend-01

Modular code, dependency mgmt, go get - #2

- create another package dog
    - https://go.dev/ref/spec#Arithmetic_operators
    - https://pkg.go.dev/strings
        - example:
            - https://go.dev/play/p/I7uqqtLshSO
- in puppy
    - import dog
    - create a new function
        - uses the dog
- in learn-to-code-go-version-03
    - inspect
        - go.mod
        - go.sum
    - go get github.com/GoesToEleven/puppy
        - go get github.com/GoesToEleven/puppy@latest
    - use puppy package again
    - inspect
        - go.mod
        - go.sum
- not capitalization being used
    - Capitalized/exported/visible & lowercase/not-exported/not-visible

https://go.dev/play/p/QJJ_2nWvKLI
curriculum item # 047-mod-code-depend-02

Tag git commits with version - overview

Git is a version control system that allows developers to track changes in their codebase and collaborate on projects with ease. One important feature of Git is the ability to tag specific versions of a project.

To tag a version in Git, follow these steps:

- Make sure you're in the branch you want to tag. You can use the command git branch to see which branch you're currently in and git checkout <branch> to switch to another branch if necessary.
- Decide on a version number for your tag. This can be any alphanumeric string, but it's common to use a format like "v1.0" or "v1.1.2".

- Use the git tag command to create a new tag. The basic syntax for creating a tag is git tag <tagname>, so if you wanted to create a tag called "v1.0", you would run git tag v1.0.
- If you want to include additional information in your tag, you can use the -a option to create an annotated tag. An annotated tag includes a message that describes the tag and can include information like who created the tag and when it was created. To create an annotated tag, use the syntax git tag -a <tagname> -m <message>.
- If you want to tag a specific commit instead of the current HEAD, use the git tag <tagname> <commit> syntax. This creates a tag at the specified commit.
- Push your tags to the remote repository using git push --tags. This will push all your local tags to the remote repository.

With these steps, you can easily create tags for specific versions of your codebase, making it easier to track changes and collaborate with others.

- two types of tags
    - lightweight
        - tags a certain commit
        - example:
          git tag v1.4
    - annotated
        - checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).
        - example:
          git tag -a v1.4 -m "my version 1.4"
            - -m specifies a tagging message, which is stored with the tag. If you don’t specify a message for an annotated tag, Git launches your editor so you can type it in.
- pushing tags
    - *By default, the git push command doesn’t transfer tags to remote servers.*
    - You have to explicitly push tags to a shared server after you have created them.
        - When you push your tags, when someone else clones or pulls from your repository, they will get all your tags as well.
    - two ways to do this
        - git push origin <tagname>.
            - example
            - git push origin v1.5
        - git push origin --tags
            - pushes all tags
            - example
            - git push origin --tags
- listing the tags (seeing all of your tags)
    - git tag (with optional -l or --list)

    - examples
    - git tag
    - git tag -l "v1.8.5*"
- for our purposes

```
git tag
git tag vN.N.N
git push origin --tags
```

- git show
    - see the tag data along with the commit that was tagged
    - example:
      git show v1.4
- semantic versioning
    - vMajor.Minor.Patch
    - Given a version number MAJOR.MINOR.PATCH, increment the:
        - MAJOR **changes, not backwards compatible**
        - MINOR **changes, backwards compatible**
        - PATCH **big fixes, backwards compatible**
- sources
    - https://git-scm.com/book/en/v2/Git-Basics-Tagging
    - https://semver.org/

curriculum item # 048-versions-overview

Tag git commits with version - example #1

For our purposes

```
git tag
git tag vN.N.N
git push origin --tags
git tag
git show v1.0.0
```

curriculum item # 049-versions-example-01

Tag git commits with version - example #2

1. Are there already tags
    a. git tag
2. add something new to our code

```
func From10(){fmt.Println("I'm from version 1.0.0")}
```

    a. commit & tag

```
git add –all
git commit -m "some commit message"
git tag v1.0.0
git push origin –tags
```

3. add something new to our code

```
func From11(){fmt.Println("I'm from version 1.2.0"}
```

   a. commit & tag

```
git add –all
git commit -m "some commit message"
git tag v1.0.0
git push origin –tags
```

4. add something new to our code

```
func From12(){fmt.Println("I'm from version 1.3.0"}
```

   a. commit & tag

```
git add –all
git commit -m "some commit message"
git tag v1.0.0
git push origin –tags
```

curriculum item # 050-versions-example-02

Specifying dependency version

- go get github.com/GoesToEleven/puppy@latest
- go get github.com/GoesToEleven/puppy@v1.1.0
- go get github.com/GoesToEleven/puppy@v1.2.0
- go get github.com/GoesToEleven/puppy@latest
- digital ocean tutorial go modules
  - https://www.digitalocean.com/community/tutorials/how-to-use-go-modules
- practical go lessons chapter 17 - go modules
  - https://www.practical-go-lessons.com/chap-17-go-modules
- Tutorial: Create a Go module

curriculum item # 051-specifying-dependency-version

Hands-on exercises

Hands-on exercise #14 (was #09)

- create the following variables with the following scopes:
  - package level
    - create outside of func main
    - use the
      - var keyword

        - const keyword
    - block level
        - inside func main
        - use the short declaration operator
- use the variables in func main

curriculum item # 052-hands-on-exercise-09

Hands-on exercise #15 (was #10)

- use the terminal to make a go workspace
    - mkdir <name>
    - cd <name>
    - go mod init <somename>
- write a hello world program
    - nano main.go
        - (if this doesn't work on your machine, use an IDE)
    - write go code
- run your program
    - go run main.go

curriculum item # 053-hands-on-exercise-10

Hands-on exercise #16 (was #11)

Using the code you wrote in the previous hands-on exercise:

- look at the contents in the folder of your module
    - ls -la
- build your program
    - any of these
        - go build main.go
        - go build .
        - go build ./…
- run your executable
    - ./<name of executable>
    - this could vary on your machine, if so, google how to do it on your machine, for example, "on a mac, run executable at the terminal" or "run executable from terminal mac"

curriculum item # 054-hands-on-exercise-11

Hands-on exercise #17 (was #12)

Using the code you wrote in the previous hands-on exercise:

- build your program for windows
    - GOOS=windows go build
- build your program for mac
    - GOOS=darwin go build
- build your program for linux
    - GOOS=linux go build

curriculum item # 055-hands-on-exercise-12

Hands-on exercise #18 (was #13)

Using the code from the previous hands-on exercise:

- look in the $GOPATH/bin
    - launch another terminal
    - see the "go path" environment variable - either of these
        - $GOPATH
        - go env
    - navigate to the $GOPATH/bin folder
    - ls -la
- "go install" your program (on the other terminal)
    - look at the executable $GOPATH/bin
    - ls -la
- run the executable in the $GOPATH/bin
- remove the executable in the $GOPATH/bin
    - if you accidentally delete everything, you will need to reinstall your tooling in VS code
    - if you messed it all up, reinstall go

curriculum item # 056-hands-on-exercise-13

Hands-on exercise #19 (was #14)

Using the code from the previous hands-on exercise:

- use a function from the package found at github.com/GoesToEleven/puppy
    - go get github.com/GoesToEleven/puppy
- inspect your go.mod file
- run go mod tidy
- what does go mod tidy do?
    - https://go.dev/ref/mod#go-mod-tidy

curriculum item # 057-hands-on-exercise-14

Hands-on exercise #20 (was #15)

Using the code from the previous hands-on exercise:

- use a function from the package found at github.com/GoesToEleven/puppy but make your code depend on v1.2.0
    - go get github.com/GoesToEleven/puppy@v1.2.0
- inspect your go.mod file

curriculum item # 058-hands-on-exercise-15

Hands-on exercise #21 (was #16)

- Create a github repo for you code
- push your code
- add a version tag to your code of v1.0.0
    - git tag
    - git tag v1.0.0
    - git push origin –tags

- add a temp.txt file to your code
- push your code
- add a version tag to your code of v1.1.0
- look at your versions in github
- optional: delete your repo

curriculum item # 059-hands-on-exercise-16

Hands-on exercise #22 & git clone (was #17)

At the terminal

- go to your go workspace where you wrote your code
- remove the folder you created to write your go code
    - rm -rf <folder name>

curriculum item # 060-hands-on-exercise-17

Housekeeping

Hash, encryption, & communication

- Hash
    - A **hash algorithm** is a mathematical function that takes in a variable-sized input, such as a file or message, and produces a fixed-size output, known as a hash or digest. The output is a unique representation of the input data, and even a small change to the input data will produce a completely different hash value. Hash algorithms are used in a variety of applications such as data integrity checks, message authentication, password storage, and digital signatures. For example, when a user sets a password, the password is hashed and stored in a database. When the user logs in, the password they enter is hashed again and compared to the stored hash to verify their identity. Hash algorithms are designed to be one-way functions, meaning it is extremely difficult (if not impossible) to reconstruct the original input data from the hash value. This makes them useful for securely storing passwords or sensitive information. Common examples of hash algorithms include MD5, SHA-1, SHA-2, and SHA-3. However, it is important to note that some of these algorithms, particularly MD5 and SHA-1, are considered insecure and should not be used for new applications.
- Encryption
    - synchronous / symmetric encryption
        - single key
    - asynchronous encryption
        - public / private key
    - Synchronous encryption and asynchronous encryption are two different approaches to encrypting data. Synchronous encryption, also known as symmetric encryption, uses a single key to both encrypt and decrypt data. This

        means that the same key is used to encrypt the data and to decrypt it. The encryption and decryption processes are fast and efficient, making synchronous encryption a popular choice for encrypting large amounts of data. Examples of synchronous encryption algorithms include Advanced Encryption Standard (AES) and Data Encryption Standard (DES). Asynchronous encryption, also known as public-key encryption, uses a pair of keys: a public key and a private key. The public key is used to encrypt the data, while the private key is used to decrypt it. As a result, anyone can encrypt the data using the public key, but only the holder of the private key can decrypt it. Asynchronous encryption is often used for secure communication over insecure channels, such as the internet. Examples of asynchronous encryption algorithms include RSA and Elliptic Curve Cryptography (ECC). Both synchronous and asynchronous encryption have their advantages and disadvantages, and the choice of which to use depends on the specific needs of the application. Synchronous encryption is typically faster and more efficient for encrypting large amounts of data, while asynchronous encryption provides better security for secure communication over insecure channels.

- communication
    - **simplex, half-duplex, full-duplex**
    - Simplex, half-duplex, and full-duplex are terms used to describe different types of communication modes in telecommunications.
    - **Simplex Communication:** In simplex communication, information flows in only one direction. This means that the sender can transmit data to the receiver, but the receiver cannot send data back to the sender. Examples of simplex communication include TV broadcasting and most remote control devices.
    - **Half-Duplex Communication:** In half-duplex communication, information can flow in both directions, but not at the same time. When one party is transmitting data, the other party must wait for the transmission to end before sending their own data. Examples of half-duplex communication include walkie-talkies and CB radios.
    - **Full-Duplex Communication:** In full-duplex communication, information can flow in both directions simultaneously. This means that both the sender and receiver can send and receive data at the same time. Examples of full-duplex communication include telephone conversations, video conferencing, and internet chat.
    - In summary, simplex communication is one-way, half-duplex communication is two-way but not at the same time, and full-duplex communication is two-way and simultaneous.

curriculum item # 061-hash-encrypt-comm

Control Flow

Previewing code

A preview of this section's code
https://go.dev/play/p/-jMpY4wg264
curriculum item # 062-previewing-code

Understanding control flow

control flow

- **sequence**
- **conditional**
- **loop**

Control flow refers to the order in which statements, instructions, and operations are executed in a computer program. It determines the sequence in which the instructions are executed and the conditions that determine whether or not certain instructions are executed. In other words, control flow governs the flow of execution in a program. It is typically controlled by conditional statements (e.g., if/else statements), loops (e.g., for loops, while loops), and function calls. These constructs allow programmers to **control the flow of their programs and make decisions based on various conditions.** For example, in an if/else statement, the program executes a certain block of code if a particular condition is met and a different block of code if the condition is not met. Similarly, in a loop, the program repeats a certain block of code until a particular condition is met. Overall, control flow is essential in programming because it enables programmers to create complex logic and algorithms that can perform a wide range of tasks.

The Go runtime system is a component of the Go programming language that manages and schedules the execution of Go programs. It includes a number of features that make it easy to write concurrent and parallel programs in Go.

The Go runtime system is responsible for the following:

- **Goroutine management:** Goroutines are lightweight threads that enable concurrent execution of Go programs. The runtime system manages the creation and destruction of goroutines, and schedules them for execution on available processors.

- **Garbage collection:** The runtime system includes a garbage collector that automatically frees memory that is no longer needed by a program. This makes it easier to write complex programs without worrying about manual memory management.

- **Memory management:** The runtime system includes a memory allocator that manages the allocation and deallocation of memory used by a program.

- **Channel management:** Channels are used in Go to communicate between goroutines. The runtime system manages the creation and destruction of channels, and ensures that messages are delivered in the correct order.

- **Stack management:** Each goroutine has its own stack, which is used to store local variables and function arguments. The runtime system manages the allocation and deallocation of stack space for each goroutine.

Overall, the Go runtime system is a key component of the Go programming language that makes it easy to write concurrent and parallel programs. It provides a number of powerful features that simplify the process of managing threads, memory, and communication between concurrent processes.

The Stack & The Heap
In Go, the stack and heap are two regions of memory used for storing variables and data during program execution.

The stack is a region of memory used for storing variables that are local to a function or a goroutine. When a function is called or a goroutine is created, a new stack frame is created on the stack to store the function arguments, local variables, and other data. The stack is a LIFO (last-in-first-out) data structure, which means that the most recently added data is the first to be removed. The stack is generally faster than the heap because it is managed automatically by the Go runtime system, and memory allocation and deallocation is relatively fast.

The heap is a region of memory used for storing variables that have a longer lifetime than those stored on the stack. When a variable is allocated on the heap, it remains there until it is explicitly deallocated by the program or until the program exits. The heap is a more flexible data structure than the stack, but it is generally slower because it requires more overhead for memory allocation and deallocation. In Go, the heap is managed automatically by the garbage collector, which frees up memory that is no longer being used by the program.

In general, Go programs try to allocate as much memory as possible on the stack because it is faster and requires less overhead. However, when a program needs to store large amounts of data or data with a longer lifetime than the stack, it must use the heap. The Go runtime system provides automatic memory management for both the stack and the heap, making it easy to write efficient and scalable concurrent programs.

The Go compiler is a program that translates Go source code into machine-readable binary code that can be executed by a computer. In other words, the Go compiler takes the human-readable code that a programmer writes in Go and converts it into instructions that a computer can understand and execute.

The Go compiler is responsible for several important tasks, including:

- Parsing: The compiler reads the source code and breaks it down into a series of tokens, which are then analyzed for syntax errors.

- Type checking: The compiler checks that the types of variables and expressions in the code are consistent and correct according to the rules of the Go language.

- Optimization: The compiler performs various optimizations to the code to make it run more efficiently on the target machine.

- Code generation: The compiler generates machine-readable binary code that can be executed by the target machine.

In Go, the standard compiler is called "gc" (short for "Go Compiler"). It is a highly optimized and efficient compiler that generates fast and efficient code.

Program execution
A complete program is created by linking a single, unimported package called the main package with all the packages it imports, transitively. The main package must have package name main and declare a function main that takes no arguments and returns no value.

```
func main() { … }
```

Program execution begins by initializing the main package and then invoking the function main. When that function invocation returns, the program exits. It does not wait for other (non-main) goroutines to complete.
https://go.dev/ref/spec#Program_execution

https://go.dev/play/p/-jMpY4wg264
curriculum item # 063-understanding-control-flow

If statements & comparison operators

An **if statement** is a programming construct used to control the flow of execution in a program based on a condition. The basic syntax of an if statement is:

```
if (condition) {
   // Code to execute if the condition is true
}
```

In this syntax, condition is an expression that evaluates to either true or false. If the condition is true, the code inside the curly braces will be executed. If the condition is false, the code inside the curly braces will be skipped and execution will continue with the next statement following the if statement.

Comparison operators compare two operands and yield an untyped boolean value.

==	equal

!=	not equal
<	less
<=	less or equal
>	greater
>=	greater or equal

https://go.dev/ref/spec#Comparison_operators

https://go.dev/play/p/UZb17ajAliq
curriculum item # 064-if-statements-comparison-operators

Understanding & using Logical operators

&&	AND
||	OR
!	NOT
true && true	true
true && false	false
true || true	true
true || false	true
!true	false

https://go.dev/ref/spec#Logical_operators
https://go.dev/play/p/J-GWBEzSAXA
https://go.dev/play/p/fqnAfOKmw0a
curriculum item # 065-logical-operators

The "statement; statement" & "comma, ok" idioms

- In an "if" statement
    - The expression may be preceded by a simple statement, which executes before the expression is evaluated.
    - https://go.dev/ref/spec#If_statements
- this is also like the "comma ok idiom"
    - (https://go.dev/doc/effective_go → use ctrl+f “comma ok”)
    - declare and assign; condition {}
        - limits scope
    - https://go.dev/play/p/OXGzjxVkag0

https://go.dev/play/p/shchvdIlFZe
curriculum item # 066-statement-statement-idiom

Using switch statements to make decisions in code

A switch statement is a control flow statement in programming that allows a program to evaluate an expression or variable and then selectively execute different blocks of code based on the value of the expression or variable. It provides a convenient way to write code that performs

different actions based on a single variable or expression without having to use multiple if/else statements. The switch statement typically consists of a single expression or variable followed by a series of case statements that specify the different values that the expression or variable can take and the corresponding blocks of code to execute in each case. If the expression or variable matches one of the specified case values, the corresponding block of code is executed, and if none of the cases match, an optional default case can be used to execute a fallback block of code.
https://go.dev/play/p/JuPB00o3YNC
curriculum item # 067-switch-statements

Using select statements for concurrency communication

- **Concurrency** and **parallelism** are related but distinct concepts in programming, including in the Go programming language.
- **Concurrency** refers to code that is written in a concurrent design pattern. This means that the code has the potential ability to execute multiple tasks simultaneously, where each task may make progress independently of the others.
    - In Go, concurrency is achieved using goroutines, lightweight threads of execution that are managed by the Go runtime.
    - A Go program can create many goroutines that run concurrently, each performing a different task.
    - The *communication and synchronization* of these goroutines is typically done using **channels**, which provide a way for goroutines to exchange data and coordinate their execution.
- **Parallelism**, on the other hand, refers to the ability of a program to execute multiple tasks simultaneously by utilizing multiple CPUs or cores.
    - Parallelism can often speed up the execution of a program by allowing multiple parts of the program to run in parallel on different processors. In Go, parallelism can be achieved by running multiple goroutines on different processors using the go keyword.
        - serial / sequential execution
            - the opposite of parallel computing
            - The opposite of code running in parallel is code running serially or sequentially. In sequential execution, each instruction or task is executed one after the other in a predefined order, so that each instruction must wait for the previous one to finish before it can start. This differs from parallel execution, where multiple instructions or tasks can be executed simultaneously. Sequential execution is typically used when the instructions or tasks are dependent on each other, or when the resources required to execute the code are limited. Parallel execution, on the other hand, is used to speed up the execution of code by running

multiple instructions or tasks at the same time, often on multiple processors or cores.

- Go makes it easy to write concurrent code using goroutines and channels.

https://go.dev/play/p/F7vL8Y63Tao
curriculum item # 068-select-statements

Understanding & using for statement to create loops

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

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

keywords

- break
- continue

https://go.dev/doc/effective_go#for
https://go.dev/play/p/tIiIP-r1wIB
curriculum item # 069-for-loop

Multiple iteration - nesting a loop within a loop

- **Nested loops** are a type of control structure used in programming to repeat a set of instructions multiple times.
- As the name suggests, nested loops consist of one loop inside another loop.
    - The outer loop is executed first,
        - and for each iteration of the outer loop, the inner loop is executed completely.
        - This means that the inner loop executes its set of instructions for every iteration of the outer loop.
- Nested loops are commonly used when working with multi-dimensional data structures such as arrays or matrices. They can also be used for tasks that require performing a specific action for every combination of two or more variables.

https://go.dev/play/p/CYL0g_6Wc8R
curriculum item # 070-nested-loops

Understanding & using for range loops

The for range loop allows iterating over the elements of an array, slice, string, map or channel.
https://go.dev/play/p/EYAvVZHGVQJ
curriculum item # 071-for-range

Finding a modulus / remainder

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

- %

In programming, the modulus (or modulo) operator is a mathematical operation that returns the remainder after division of one number by another. The modulus operator is denoted by the percent sign (%).

For example, in the expression 11 % 3, the modulus operator would return 2, because 11 divided by 3 leaves a remainder of 2.

The modulus operator is commonly used in programming for tasks such as checking if a number is even or odd, or for calculating the position of elements in an array.

https://go.dev/play/p/OeOR10EATed
curriculum item # 072-modulus

Hands-on exercises

Hands-on exercise #23 (was #18)

- This hands-on exercise has a text file associated with it.
- When I ran a SHA-256 checksum on this text file, I got this hash:
    - 7c6c8937b2a120af15849db05c9f46326761e0eec852a2e973b1e0b6acd59a01
- Download the text file associated with this hands-on exercise. Run a SHA-256 checksum on it. Do you get the same hash?
    - shasum -a 256 /path/to/file
        - . is the current directory
        - if you're in the directory with the file
            - shasum -a 256 ./file
- change the file by one character, then run SHA-256 again
    - 9be13f9173f28ce3dd89c72aad7f5b0549a0641feb869509c7f96e8dc8b6ea8e

ADD TEXT FILE
curriculum item # 073-hands-on-exercise-18

Hands-on exercise #24 (was #19)

- create a program that uses both SEQUENTIAL and CONDITIONAL control flow. Your program should do the following
    - create a random int between 0 and 250
    - store the value of that int in a variable with the identifier of x
        - func Intn(n int) int
            - rand.Intn()
    - print out the name and value of the variable

        - use an if statement to do the following
            - if the value is between 0 and 100
                - print between 0 and 100
            - if the value is between 101 and 200
                - print between 101 and 200
            - if the value is between 201 and 250
                - print between 201 and 250
- re: inclusive, exclusive – does rand.Intn()
    - include zero in its output?
    - include 250 in its output?
    - show this in code using the numbers 0 - 3
- hint:
    - &&

curriculum item # 074-hands-on-exercise-19

Hands-on exercise #25 (was #20)

- Modify the previous program to use one of these conditional logic statements
    - a switch statement
    - a select statement
- Which of the above conditional logic statements did you choose and why?

curriculum item # 075-hands-on-exercise-20

Hands-on exercise #26 (was #21)

- Modify the previous program to have your program use the init func to print
    - "This is where initialization for my program occurs"
- read about the init function in effective go
    - What does niladic mean?

curriculum item # 076-hands-on-exercise-21

Hands-on exercise #27 (was #22)

- Create 2 random ints between 0 inclusive and 10 exclusive
    - assign them to variables with the identifiers x and y
- Print their values
- use an if statement to print the correct description
    - x and y are both less than 4
    - x and y are both greater than 6
    - x is greater than or equal to 4 and less than or equal to 6
    - y is not 5
    - none of the previous cases were met

curriculum item # 077-hands-on-exercise-22

Hands-on exercise #28 (was #23)

- Modify the previous program to use a switch statement

curriculum item # 078-hands-on-exercise-23

Hands-on exercise #29 (was #24)

- there are two parts ot this hands on exercise
    - create a program that has a loop that prints every number from 0 to 99
    - modify the program from the previous hands on exercise to run 100 times

curriculum item # 079-hands-on-exercise-24

Hands-on exercise #30 (was #25)

- Create one random int between 0 inclusive and 5 exclusive
    - assign the value to a variable with the identifier x
- Use a switch statement to print a description of the variable and value
- run the code 42 times and print the iteration number

curriculum item # 080-hands-on-exercise-25

Hands-on exercise #31 (was #26) & infinite loops

- create a for loop using only a condition
- increment or decrement the variable being checked in the condition

curriculum item # 081-hands-on-exercise-26

Hands-on exercise #32 (was #27)

- create a for loop that uses **break** statement
- increment or decrement the variable being checked as a condition in the loop

curriculum item # 082-hands-on-exercise-27

Hands-on exercise #33 (was #28) & a joke

- use modulus and the **continue** statement in a loop to print all ODD numbers
- joke about the programmer and infinite loops

curriculum item # 083-hands-on-exercise-28

Hands-on exercise #34 (was #29)

- create a loop that runs 5 times
- nest a loop within the first loop that also prints 5 times
- print something in each loop to illustrate what is occuring

curriculum item # 084-hands-on-exercise-29

Hands-on exercise #35 (was #30)

- below is the code to create a data structure called a slice of ints
- put this code into a program

```
xi := []int{42, 43, 44, 45, 46, 47}
```

- use a **for range loop** to print each **value** and the **index** position of each value
  😃🌴🌞🔆🌻🙌🌴🌴🌴🌴🌴🌴🌴🌴

curriculum item # 085-hands-on-exercise-30

Hands-on exercise #36 (was #31)

- below is the code to create a data structure called a map
- put this code into a program

```
m := map[string]int{
        "James":      42,
        "Moneypenny": 32,
}
```

- use a **for range loop** to print each **value** and the **key** associated with each value

curriculum item # 086-hands-on-exercise-31

Hands-on exercise #37 (was #32)

- use the code from the previous exercise
- add this code to print a single value stored in the map

```
age := m["James"]
fmt.Println(age)
```

- now use similar code to use the lookup of "Q" and print that value
- now use the "**comma ok**" idiom to test whether "Q" is actually stored in the map, then print out a statement if it is not stored in the map
  - hint: check effective go for the "comma ok" idiom

curriculum item # 087-hands-on-exercise-32

Hands-on exercise #38 (was #33)

- use the "statement statement" idiom to
  - initialize x with and random int between 0 inclusive and 5 exclusive
  - if x is 3
    - print "x is 3"
- run that code 100 times
- what's the benefit of using the "statement statement" idiom?

curriculum item # 088-hands-on-exercise-33

Hands-on exercise #39 (was #34)

What do these print:

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

curriculum item # 089-hands-on-exercise-34

Additional code

curriculum item # 090-additional-code

Grouping data values - array & slice

Review and preview

curriculum item # 091-review-preview

Introduction to grouping values

- **array**
  - **a numbered sequence of VALUES of the same TYPE**
  - does not change in size
  - **used for Go internals; generally not used in 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 VALUES 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 / aggregate

- allows us to **collect values of different types together**
- https://golang.org/ref/spec#Struct_types

curriculum item # 092-intro-grouping-data

Array - an introduction to arrays

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://go.dev/play/p/_tZD1mCUjej
- https://play.golang.org/p/f-7aufl2DO

curriculum item # 093-intro-to-arrays

Hands-on exercise #40

- Use an **array literal** to store these elements in an array and let the compiler determine the length of the array, then also print out the array and the **length** of the array

  "Almond Biscotti Café", "Banana Pudding ", "Balsamic Strawberry (GF)", "Bittersweet Chocolate (GF)", "Blueberry Pancake (GF)", "Bourbon Turtle (GF)", "Browned Butter Cookie Dough", "Chocolate Covered Black Cherry (GF)", "Chocolate Fudge Brownie", "Chocolate Peanut Butter (GF)", "Coffee (GF)", "Cookies & Cream", "Fresh Basil (GF)", "Garden Mint Chip (GF)", "Lavender Lemon Honey (GF)", "Lemon Bar", "Madagascar Vanilla (GF)", "Matcha (GF)", "Midnight Chocolate Sorbet (GF, V)", "Non-Dairy Chocolate Peanut Butter (GF, V)", "Non-Dairy Coconut Matcha (GF, V)", "Non-Dairy Sunbutter Cinnamon (GF, V)", "Orange Cream (GF) ", "Peanut Butter Cookie Dough", "Raspberry Sorbet (GF, V)", "Salty Caramel (GF)", "Slate Slate Different", "Strawberry Lemonade Sorbet (GF, V)", "Vanilla Caramel Blondie", "Vietnamese Cinnamon (GF)", "Wolverine Tracks (GF)"

https://go.dev/play/p/OEHPUGDoWD_t

curriculum item # 094-hands-on-exercise-40

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 SLICE LITERAL to create a slice.** A slice literal is created by having the TYPE followed by CURLY BRACES and then putting the appropriate values in the curly brace area.

code

- https://go.dev/play/p/5woDCfR2LLi
- https://play.golang.org/p/XtUEPJFgqD

curriculum item # 095-slice-composite-literal

Hands-on exercise #41

- Use a **slice literal** to store these elements in a slice, then also print out the slice and the **length** of the slice.
- if you can, try to get a "for range" loop working on the slice
    - you can reference the documentation for a "for range" loop here:
        - https://go.dev/doc/effective_go#for

    "Almond Biscotti Café", "Banana Pudding ", "Balsamic Strawberry (GF)", "Bittersweet Chocolate (GF)", "Blueberry Pancake (GF)", "Bourbon Turtle (GF)", "Browned Butter Cookie Dough", "Chocolate Covered Black Cherry (GF)", "Chocolate Fudge Brownie", "Chocolate Peanut Butter (GF)", "Coffee (GF)", "Cookies & Cream", "Fresh Basil (GF)", "Garden Mint Chip (GF)", "Lavender Lemon Honey (GF)", "Lemon Bar", "Madagascar Vanilla (GF)", "Matcha (GF)", "Midnight Chocolate Sorbet (GF, V)", "Non-Dairy Chocolate Peanut Butter (GF, V)", "Non-Dairy Coconut Matcha (GF, V)", "Non-Dairy Sunbutter Cinnamon (GF, V)", "Orange Cream (GF) ", "Peanut Butter Cookie Dough", "Raspberry Sorbet (GF, V)", "Salty Caramel (GF)", "Slate Slate Different", "Strawberry Lemonade Sorbet (GF, V)", "Vanilla Caramel Blondie", "Vietnamese Cinnamon (GF)", "Wolverine Tracks (GF)"

https://go.dev/play/p/W8sWn7fsUXh
curriculum item # 096-hands-on-exercise-41

Slice - for range & access values by index position

- We can **loop over the values in a slice with the range clause.**
    - blank identifier
- We can also access items in a slice by index position.
- length of slice with len function
- using a traditional for loop to access values of a slice by index position

code

- https://go.dev/play/p/YQ89CoNq04G
- https://play.golang.org/p/O7cCALNFsH

curriculum item # 097-for-range-slice

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://go.dev/play/p/1PgUc0KvaTX
- https://play.golang.org/p/oQnjP5Ka3F

curriculum item # 098-append-to-slice

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://go.dev/play/p/pYutmmXO8Q9
- https://play.golang.org/p/AXGTEEn92M

curriculum item # 99-slicing-a-slice

Slice - deleting from a slice

We can **delete from a slice using both append and slicing.**

code:

https://go.dev/play/p/qm_RV-xnoeA

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

curriculum item # 100-delete-from-slice

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://go.dev/play/p/CwfCyWF3Tei
- https://play.golang.org/p/07hH1b-hvD

curriculum item # 101-slice-make

Slice - multidimensional 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://go.dev/play/p/BjvYAIcUjCK
- https://play.golang.org/p/S4cyB89Zpm

curriculum item # 102-multidimensional-slice

Slice - slice internals & underlying array - 01

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

https://go.dev/play/p/YXoRT0wqz9F
https://cs.opensource.google/go/go/+/refs/tags/go1.20.4:src/runtime/slice.go
curriculum item # 103-slice-internals-01

Slice - slice internals & underlying array - 02

You can create a new slice, with a new underlying array, using the builtin func "make"
https://go.dev/play/p/JgnnzAYB4Y8
curriculum item # 104-slice-internals-02

Slice - slice internals & underlying array - 03

Another example of thinking about the underlying array when passing slices to functions
https://go.dev/play/p/ZBz2jFnL8oc
curriculum item # 105-slice-internals-03

Hands-on exercises

Hands-on exercise #42

- 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.
    - print out the VALUE and the TYPE

https://go.dev/play/p/aghCIHXIlsB
curriculum item # 106-hands-on-exercise-42

Hands-on exercise #43

- Using a COMPOSITE LITERAL:
    - create a SLICE of TYPE int
    - assign these 10 VALUES
      42, 43, 44, 45, 46, 47, 48, 49, 50, 51
- Range over the slice and print the values out.
    - print out the VALUE and the TYPE

https://go.dev/play/p/84r6QDOaMrs
curriculum item # 107-hands-on-exercise-43

Hands-on exercise #44

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]

https://go.dev/play/p/LoAQoGHT-W6
curriculum item # 108-hands-on-exercise-44

Hands-on exercise #45

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

https://go.dev/play/p/ImPDBWxI8Mo
curriculum item # 109-hands-on-exercise-45

Hands-on exercise #46

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 the variable “x” and then print:
    - [42, 43, 44, 48, 49, 50, 51]

https://go.dev/play/p/DWR7qRa-VIR
curriculum item # 110-hands-on-exercise-46

Hands-on exercise #47

For this exercise, do the following:

- 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.
- Print out
  - the len
  - the cap
  - 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`,
https://go.dev/play/p/-cwlxRpnJAw
curriculum item # 111-hands-on-exercise-47

Hands-on exercise #48

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", "I'm 008."

Range over the records, then range over the data in each record.
https://go.dev/play/p/2jxGMyXiZrE
curriculum item # 112-hands-on-exercise-48

Grouping data values - map

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 a friend's name “Daniel” as a key and then have the value of a phone number "777-888-9999" as the value. 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. A map is the perfect data structure when you need to look up data fast.

- creating a map
- add an element to your map
- accessing elements in a map
- printing out the entire map
- seeing the length of the map
  - len
  - https://pkg.go.dev/builtin#len

https://go.dev/play/p/OmgfgoUxsgV
curriculum item # 113-map-intro

Map - for range over a map

Here is how we **use the range loop to print out a map's** keys and values.

- range over a map
- range of a slice

https://go.dev/play/p/tNQNtNrubNL
curriculum item # 114-map-for-range

Map - delete element

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.
https://go.dev/play/p/TbffT21Ljqj
curriculum item # 115-map-delete-element

Map - comma ok idiom

If you look up a non-existent key, the zero value will be returned as the value associated with that non-existent key. You can check to see if you have looked up an existing key, or a non-existent key, using the comma ok idiom
https://go.dev/play/p/lRcjXnOupDE
curriculum item # 116-map-comma-ok-idiom

Map - counting words in a book

This is a preview of some more advanced code, but core to this code is using a map to count the words in a book.
https://go.dev/play/p/AEC4DhAVmpV
curriculum item # 117-map-count-words

Hands-on exercises

Hands-on exercise #49 - map[string][]string

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.

key	**value**
\`bond_james\`	\`shaken, not stirred\`, \`martinis\`, \`fast cars\`
\`moneypenny_jenny\`	\`intelligence\`, \`literature\`, \`computer science\`
\`no_dr\`	\`cats\`, \`ice cream\`, \`sunsets\`

https://go.dev/play/p/nAqHIjP8JTZ
curriculum item # 118-hands-on-exercise-49

Hands-on exercise #50 - add a record

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

key	**value**
\`fleming_ian\`	\`steaks\`, \`cigars\`, \`espionage\`

https://go.dev/play/p/AJ8ZsHCC0wU
curriculum item # 119-hands-on-exercise-50

Hands-on exercise #51 - delete a record

Using the code from the previous example, delete a record from your map. Now print the map out using the "range" loop
https://go.dev/play/p/yba3zekRiBY
curriculum item # 120-hands-on-exercise-51

Hands-on exercise #52 - word frequency

Use a for range loop to loop over these words, then count how many times each word occurs.

The first 1000 words from **The Great Gatsby**

```
"in", "my", "younger", "and", "more", "vulnerable", "years", "my", "father", "gave", "me", "some",
"advice", "that", "i’ve", "been", "turning", "over", "in", "my", "mind", "ever", "since.", "whenever",
"you", "feel", "like", "criticizing", "anyone,", "he", "told", "me,", "just", "remember", "that", "all",
"the", "people", "in", "this", "world", "haven’t", "had", "the", "advantages", "that", "you’ve",
"had.", "he", "didn’t", "say", "any", "more,", "but", "we’ve", "always", "been", "unusually",
"communicative", "in", "a", "reserved", "way,", "and", "i", "understood", "that", "he", "meant",
"a", "great", "deal", "more", "than", "that.", "in", "consequence,", "i’m", "inclined", "to",
"reserve", "all", "judgements,", "a", "habit", "that", "has", "opened", "up", "many", "curious",
"natures", "to", "me", "and", "also", "made", "me", "the", "victim", "of", "not", "a", "few",
"veteran", "bores.", "the", "abnormal", "mind", "is", "quick", "to", "detect", "and", "attach",
"itself", "to", "this", "quality", "when", "it", "appears", "in", "a", "normal", "person,", "and", "so",
"it", "came", "about", "that", "in", "college", "i", "was", "unjustly", "accused", "of", "being", "a",
"politician,", "because", "i", "was", "privy", "to", "the", "secret", "griefs", "of", "wild,", "unknown",
"men.", "most", "of", "the", "confidences", "were", "unsought—frequently", "i", "have",
"feigned", "sleep,", "preoccupation,", "or", "a", "hostile", "levity", "when", "i", "realized", "by",
"some", "unmistakable", "sign", "that", "an", "intimate", "revelation", "was", "quivering", "on",
"the", "horizon;", "for", "the", "intimate", "revelations", "of", "young", "men,", "or", "at", "least",
"the", "terms", "in", "which", "they", "express", "them,", "are", "usually", "plagiaristic", "and",
"marred", "by", "obvious", "suppressions.", "reserving", "judgements", "is", "a", "matter", "of",
"infinite", "hope.", "i", "am", "still", "a", "little", "afraid", "of", "missing", "something", "if", "i",
"forget", "that,", "as", "my", "father", "snobbishly", "suggested,", "and", "i", "snobbishly",
"repeat,", "a", "sense", "of", "the", "fundamental", "decencies", "is", "parcelled", "out",
"unequally", "at", "birth.", "and,", "after", "boasting", "this", "way", "of", "my", "tolerance,", "i",
"come", "to", "the", "admission", "that", "it", "has", "a", "limit.", "conduct", "may", "be",
"founded", "on", "the", "hard", "rock", "or", "the", "wet", "marshes,", "but", "after", "a", "certain",
"point", "i", "don’t", "care", "what", "it’s", "founded", "on.", "when", "i", "came", "back", "from",
"the", "east", "last", "autumn", "i", "felt", "that", "i", "wanted", "the", "world", "to", "be", "in",
"uniform", "and", "at", "a", "sort", "of", "moral", "attention", "forever;", "i", "wanted", "no",
"more", "riotous", "excursions", "with", "privileged", "glimpses", "into", "the", "human", "heart.",
"only", "gatsby,", "the", "man", "who", "gives", "his", "name", "to", "this", "book,", "was",
"exempt", "from", "my", "reaction—gatsby,", "who", "represented", "everything", "for", "which",
"i", "have", "an", "unaffected", "scorn.", "if", "personality", "is", "an", "unbroken", "series", "of",
"successful", "gestures,", "then", "there", "was", "something", "gorgeous", "about", "him,",
"some", "heightened", "sensitivity", "to", "the", "promises", "of", "life,", "as", "if", "he", "were",
"related", "to", "one", "of", "those", "intricate", "machines", "that", "register", "earthquakes",
"ten", "thousand", "miles", "away.", "this", "responsiveness", "had", "nothing", "to", "do", "with",
"that", "flabby", "impressionability", "which", "is", "dignified", "under", "the", "name", "of", "the",
"creative", "temperament—it", "was", "an", "extraordinary", "gift", "for", "hope,", "a", "romantic",
"readiness", "such", "as", "i", "have", "never", "found", "in", "any", "other", "person", "and",
"which", "it", "is", "not", "likely", "i", "shall", "ever", "find", "again.", "no—gatsby", "turned", "out",
"all", "right", "at", "the", "end;", "it", "is", "what", "preyed", "on", "gatsby,", "what", "foul", "dust",
"floated", "in", "the", "wake", "of", "his", "dreams", "that", "temporarily", "closed", "out", "my",
"interest", "in", "the", "abortive", "sorrows", "and", "short-winded", "elations", "of", "men.", "my",
"family", "have", "been", "prominent,", "well-to-do", "people", "in", "this", "middle", "western",
"city", "for", "three", "generations.", "the", "carraways", "are", "something", "of", "a", "clan,",
"and", "we", "have", "a", "tradition", "that", "we’re", "descended", "from", "the", "dukes", "of",
"buccleuch,", "but", "the", "actual", "founder", "of", "my", "line", "was", "my", "grandfather’s",
"brother,", "who", "came", "here", "in", "fifty-one,", "sent", "a", "substitute", "to", "the", "civil",
"war,", "and", "started", "the", "wholesale", "hardware", "business", "that", "my", "father",
"carries", "on", "today.", "i", "never", "saw", "this", "great-uncle,", "but", "i’m", "supposed", "to",
"look", "like", "him—with", "special", "reference", "to", "the", "rather", "hard-boiled", "painting",
"that", "hangs", "in", "father’s", "office.", "i", "graduated", "from", "new", "haven", "in", "1915,",
"just", "a", "quarter", "of", "a", "century", "after", "my", "father,", "and", "a", "little", "later", "i",
"participated", "in", "that", "delayed", "teutonic", "migration", "known", "as", "the", "great",
"war.", "i", "enjoyed", "the", "counter-raid", "so", "thoroughly", "that", "i", "came", "back",
"restless.", "instead", "of", "being", "the", "warm", "centre", "of", "the", "world,", "the", "middle",
"west", "now", "seemed", "like", "the", "ragged", "edge", "of", "the", "universe—so", "i",
"decided", "to", "go", "east", "and", "learn", "the", "bond", "business.", "everybody", "i", "knew",
"was", "in", "the", "bond", "business,", "so", "i", "supposed", "it", "could", "support", "one",
"more", "single", "man.", "all", "my", "aunts", "and", "uncles", "talked", "it", "over", "as", "if",
"they", "were", "choosing", "a", "prep", "school", "for", "me,", "and", "finally", "said,",
"why—ye-es,", "with", "very", "grave,", "hesitant", "faces.", "father", "agreed", "to", "finance",
"me", "for", "a", "year,", "and", "after", "various", "delays", "i", "came", "east,", "permanently,",
"i", "thought,", "in", "the", "spring", "of", "twenty-two.", "the", "practical", "thing", "was", "to",
"find", "rooms", "in", "the", "city,", "but", "it", "was", "a", "warm", "season,", "and", "i", "had",
"just", "left", "a", "country", "of", "wide", "lawns", "and", "friendly", "trees,", "so", "when", "a",
"young", "man", "at", "the", "office", "suggested", "that", "we", "take", "a", "house", "together",
"in", "a", "commuting", "town,", "it", "sounded", "like", "a", "great", "idea.", "he", "found", "the",
"house,", "a", "weather-beaten", "cardboard", "bungalow", "at", "eighty", "a", "month,", "but",
"at", "the", "last", "minute", "the", "firm", "ordered", "him", "to", "washington,", "and", "i", "went",
"out", "to", "the", "country", "alone.", "i", "had", "a", "dog—at", "least", "i", "had", "him", "for", "a",
"few", "days", "until", "he", "ran", "away—and", "an", "old", "dodge", "and", "a", "finnish",
"woman,", "who", "made", "my", "bed", "and", "cooked", "breakfast", "and", "muttered",
"finnish", "wisdom", "to", "herself", "over", "the", "electric", "stove.", "it", "was", "lonely", "for",
"a", "day", "or", "so", "until", "one", "morning", "some", "man,", "more", "recently", "arrived",
"than", "i,", "stopped", "me", "on", "the", "road.", "how", "do", "you", "get", "to", "west", "egg",
"village?", "he", "asked", "helplessly.", "i", "told", "him.", "and", "as", "i", "walked", "on", "i",
"was", "lonely", "no", "longer.", "i", "was", "a", "guide,", "a", "pathfinder,", "an", "original",
"settler.", "he", "had", "casually", "conferred", "on", "me", "the", "freedom", "of", "the",
"neighbourhood.", "and", "so", "with", "the", "sunshine", "and", "the", "great", "bursts", "of",
"leaves", "growing", "on", "the", "trees,", "just", "as", "things", "grow", "in", "fast", "movies,", "i",
"had", "that", "familiar", "conviction", "that", "life", "was", "beginning", "over", "again", "with",
"the", "summer.", "there", "was", "so", "much", "to", "read,", "for", "one", "thing,", "and", "so",
"much", "fine", "health", "to", "be", "pulled", "down", "out", "of", "the", "young", "breath-giving",
"air.", "i", "bought", "a", "dozen", "volumes", "on", "banking", "and", "credit", "and",
"investment", "securities,", "and", "they", "stood", "on", "my", "shelf", "in", "red"}
```

https://go.dev/play/p/3quGhpVWGg2
curriculum item # 121-hands-on-exercise-52

Grouping data values - structs

Struct introduction

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.**
https://go.dev/play/p/3NnHshCydm5
curriculum item # 122-intro-to-structs

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.
https://go.dev/play/p/w8WBKiKhGX4
curriculum item # 123-embedded-structs

Anonymous structs

We can create anonymous structs also. **An anonymous struct is a struct which is not associated with a specific identifier.**
https://go.dev/play/p/o-csXhWFnFH
curriculum item # 124-anonymous-structs

Composition

composite *data types, aka,* ***aggregate*** *data types, aka,* ***complex*** *data types*

In Go, **composition** refers to a way of structuring and building complex types by combining multiple simpler types. Composition is one of the fundamental principles of object-oriented programming and allows you to create more flexible and reusable code.

One of the ways composition is achieved is by **embedding one struct type within another.** The fields and methods of the embedded "inner" struct become accessible to the outer struct. The inner type is said to be **promoted** to the outer type. In addition, methods of the inner type are also **promoted** to the outer type. This concept is similar to inheritance in traditional object-oriented languages, but Go does not have a built-in inheritance mechanism.

Here's an example to illustrate composition in Go:

```
type Engine struct {
    // Engine fields
}

// engine method
func (e *Engine) Start() {
    fmt.Println("Engine started")
}

type Car struct {
    Engine  // Embedding the Engine struct
    // Car-specific fields
}

func main() {
    car := Car{}
    car.Start() // Calling the Start() method from the embedded Engine struct
}
```

In the example above, we have two struct types: `Engine` and `Car`. The `Car` struct embeds the `Engine` struct, which means it includes all the fields and methods of the `Engine` struct within itself. This allows the `Car` struct to access and use the `Start()` method of the embedded `Engine` struct.

By using **composition**, you can build complex types by combining simpler ones, promoting code reuse and modular design. It provides a way to create relationships between different types without the need for traditional inheritance.

- 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**.
- 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
    - why create a type?
        - if you need to attach methods to a type
        - can also self-document your code
        - see the time package for an example
            - https://pkg.go.dev/time#Duration
            - type Duration int64
                - Duration explains what it is
                - 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, int64, float32, bool

curriculum item # 125-composition

Hands-on exercises

Hands-on exercise #53 - struct with slice

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.
https://go.dev/play/p/nLcea3bIb7h
curriculum item # 126-hands-on-exercise-53-struct-with-slice

Hands-on exercise #54 - map struct

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.
https://go.dev/play/p/-9q96CysQfG
curriculum item # 127-hands-on-exercise-54-map-struct

Hands-on exercise #55 - embed struct

- Create a type engine struct, and include this field
    - electric bool
- Create a type vehicle struct, and include these fields
        - engine
        - make
        - model
        - doors
        - color
- Create two VALUES of TYPE vehicle
    - use a composite literal
- Print out each of these values.
- Print out a single field from each of these values.

https://go.dev/play/p/6e6aua0fgSR

curriculum item # 128-hands-on-exercise-55-embed-struct

Hands-on exercise #56 - anonymous struct

Create and use an anonymous struct with these fields:

- first   string
- friends   map[string]int
- favDrinks []string

Print things.
https://go.dev/play/p/I72ejdKEe3r
curriculum item # 129-hands-on-exercise-56-anon-struct

Functions in the go programming language

Introduction to functions

- introduction to modular code
    - spaghetti code
    - structured / procedural programming
        - purpose
            - "abstract" code
            - code reusability
            - more understandable
        - functions
        - packages

In Go programming language, a function is a group of statements that together perform a specific task. It is an important concept as it allows us to create reusable and modular code.

Every Go program has at least one function, which is the `main()` function. The execution of a Go program starts and ends with this function.

Here's the basic syntax of a function in Go:

```go
func functionname(parametername type) returntype {
   //function body
}
```

Breakdown of the syntax:

1. `func`: This is the keyword to start the declaration of a function.

2. `functionname`: This is the name of the function. The function name and the parameter list together constitute the function signature.

3. `parametername type`: These are the parameters accepted by the function. You can provide any number of parameters separated by a comma. Each parameter has a name and a type.

4. `returntype`: This is the data type of the value the function returns. If a function doesn't return any value, the return type can be omitted.

5. `{ function body }`: The function body is enclosed between `{}`. The function body can contain statements and also a return statement to return a value from the function.

For example:

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

func main() {
    result := add(10, 20)
    fmt.Println(result)
}
```

In this example, we've defined a function named `add` that takes two integer parameters and returns an integer. It adds the two numbers and returns the result. In the `main()` function, we call the `add` function with two numbers, and then print the result.

Go supports multiple return values from a function. Here's how:

```go
func rectangle(length, width int) (int, int) {
    var area = length * width
    var perimeter = (length + width) * 2
    return area, perimeter
}

func main() {
    area, perimeter := rectangle(10, 20)
    fmt.Println("Area: ", area)
    fmt.Println("Perimeter: ", perimeter)
}
```

~~~~
```
~~~~

In this example, the `rectangle` function calculates the area and perimeter of a rectangle, and returns them. In the `main()` function, we call the `rectangle` function and print the returned area and perimeter.

There's a lot more to functions in Go, like variadic functions, function literals (also known as anonymous functions), higher-order functions, and more. However, this should give you a good starting point for understanding functions in Go.

curriculum item # 130-func-introduction

Syntax of functions in Go

- **func (receiver) identifier(parameters) (returns) { code }**
    - parameters & 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**
- sample funcs
    - no params, no returns
    - 1 param, no returns
    - 1 param, 1 return
    - 2 params, 2 returns

https://go.dev/play/p/VXAaDKtgVAq
curriculum item # 131-func-syntax

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).

- create a func that takes an unlimited number of VALUES of TYPE int

A variadic parameter in Go programming language is a parameter that can take an arbitrary number of arguments of a particular type. It's a way of allowing a function to accept as many arguments as needed.

You declare a variadic function with an ellipsis `...` before the type of the last parameter. Here's an example:

~~~~
```go
func sum(nums ...int) int {
   total := 0
~~~~

~~~
    for _, num := range nums {
        total += num
    }
    return total
}
```
~~~

In the `sum` function, `nums` is a variadic parameter of type `int`. The function can be called with any number of `int` arguments:

```go
sum(1, 2)
sum(1, 2, 3)
sum(1, 2, 3, 4)
```

Inside the function, `nums` is an `[]int` (slice of `int`). The `range` keyword is used to iterate over this slice.

https://go.dev/play/p/N_VWQgL0u0o
curriculum item # 132-variadic-parameter

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.**
When you pass a slice of type T into a function that has a variadic parameter of type T in Go, you're essentially passing a variable number of arguments of the same type to the function. In Go, this feature is called "variadic functions".

However, it's important to note that you can't directly pass a slice to a variadic function. You need to use the ellipsis (...) operator to unpack the slice, as shown in this example:

~~~
```go
func Sum(nums ...int) int {
    sum := 0
    for _, num := range nums {
        sum += num
    }
    return sum
}

func main() {
~~~

```go
    nums := []int{1, 2, 3, 4}
    fmt.Println(Sum(nums...))  // here nums slice is unpacked using ...
}
```

In the example above, `nums...` is the use of the ellipsis operator to unpack the `nums` slice into a number of arguments to match the `nums ...int` parameter in the `Sum` function. This is the mechanism used to pass a slice to a variadic function in Go.

https://go.dev/play/p/10tDA0_aswX
curriculum item # 133-unfurling-a-slice

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**.

In Go programming language, the `defer` keyword is used to ensure that a function call is executed later in a program's execution, usually for purposes of cleanup. `defer` is often used where `ensure` and `finally` would be used in other languages.

When a function call is deferred, it's placed onto a stack. The function calls on the stack are executed when the surrounding function returns, not when the `defer` statement is called. They're executed in Last In, First Out (LIFO) order, so if you have multiple `defer` statements, the most recently deferred function will be the first one to execute when the function returns.

Here's a simple example:

```go
package main

import "fmt"

func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
}
```

In this example, the output will be:

~~~
```
hello
world
```
~~~

Despite the "world" line being deferred before "hello" is printed, it only actually gets printed after the `main` function has finished executing all its other statements, including printing "hello".

This makes `defer` particularly useful for handling things like closing files or releasing resources. For instance, you can open a file, defer the `Close()` operation, and then proceed with reading from or writing to the file, confident in the knowledge that it will be closed properly when the function is done, no matter what happens:

~~~
```go
package main

import (
    "fmt"
    "os"
)

func main() {
    f, err := os.Open("myfile.txt")
    if err != nil {
        fmt.Println(err)
        return
    }

    defer f.Close()

    // continue to work with file...
}
```
~~~

In the above code, even if an error occurs while working with the file, the `defer f.Close()` statement ensures that the file will be closed before the function returns.

https://go.dev/play/p/x_abhRmKDdx
curriculum item # 134-defer

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.
Methods in the Go programming language are essentially functions, but they're associated with a specific type. The type they are associated with is called the receiver type, and the method can be called using any variable of that type.

Here's the basic syntax for defining a method in Go:

```go
func (receiver ReceiverType) MethodName(parameters) ReturnType {
    // method body
}
```

In this syntax:

- `func` is the keyword that starts a function declaration.
- `receiver` is a variable that you will use to access the data structure in the method. It is effectively an argument of the method.
- `ReceiverType` is the type that this method is associated with. It can be any type except for interface types or pointer types.
- `MethodName` is the name of the method. It follows the same conventions as naming functions in Go.
- `parameters` and `ReturnType` work just like in a function declaration. The parameters are input, and the return type is what the function will output.

Here is an example of a method in Go. Assume we have a struct type `Circle`, with a method `Area`:

```go
type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}
```

In this example, `Area` is a method with receiver type `Circle`. You can call it like this:

```go
c := Circle{Radius: 5}
fmt.Println(c.Area())  // Output: 78.53981633974483
```

In this case, `c` is an instance of type `Circle`, and `Area` is a method that can be called on instances of `Circle`.

Note: If a method needs to modify the receiver or the receiver is large, it's common to use a pointer to the receiver type instead:

```go
func (c *Circle) Scale(factor float64) {
    c.Radius *= factor
}
```

In this case, `Scale` is a method with receiver type `*Circle` (pointer to Circle). It modifies the `Radius` of the circle it is called on. The method can be called as follows:

```go
c := &Circle{Radius: 5}
c.Scale(2)
fmt.Println(c.Radius)  // Output: 10
```

This shows that the `Radius` of `c` was indeed modified by the `Scale` method.

https://go.dev/play/p/1bCH57BFVpD
curriculum item # 135-methods

Interfaces & polymorphism

An **interface** in Go defines a set of method signatures.
Polymorphism is the ability of a VALUE of a certain TYPE to also be of another TYPE.
In Go, values can be of more than one type.

Polymorphism refers to the ability of an object to be of an additional TYPE and take on different behaviors. In the context of programming, it allows VALUES of different TYPES to be treated as instances of a common TYPE.

Imagine you have TYPES such as "Dog", "Cat", and "Bird"

You can make an interface TYPE called "Animal" that requires a method called "MakeSound()" to implement this TYPE.

Now, you can have TYPES "Dog", "Cat", and "Bird" define their own implementation of the method "MakeSound()" and any VALUES of TYPE "Dog", "Cat", and "Bird" will also be of TYPE "Animal"

When you call the "MakeSound()" method for any VALUE of TYPE "Animal", the specific implementation for the underlying TYPE ("Dog", "Cat", or "Bird") will be invoked. This behavior is known as polymorphism.

In simpler terms, polymorphism allows you to treat different VALUES of different TYPES as the same interface TYPE when they share a common interface. This enables you to write code that can operate on a variety of TYPES without needing to know their exact underlying TYPE, as long as they adhere to the shared behavior defined by the interface 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{} or any), then that TYPE ***implicitly implements*** the interface and is ***also*** of that interface type.

In Go, values can be of more than one type.

1. Interfaces:
An interface in Go defines a set of method signatures. It describes the behavior or contract that a type should adhere to. Any type that implements the methods specified in an interface is said to satisfy that interface. In Go, interfaces are i**mplicitly implemented;** there's no need to explicitly declare that a type implements an interface.

Here's an example of an interface declaration in Go:

```
type Shape interface {
   Area() float64
   Perimeter() float64
}
```

The `Shape` interface has two methods: `Area()` and `Perimeter()`, both of which return `float64`. Any type that defines these two methods will implicitly satisfy the `Shape` interface. For instance, if we have a `Circle` type with `Area()` and `Perimeter()` methods, it satisfies the

`Shape` interface.

```
type Circle struct {
    radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c Circle) Perimeter() float64 {
    return 2 * math.Pi * c.radius
}
```

Now, if we create a variable of the `Shape` type and assign a `Circle` instance to it, it is valid because `Circle` implements the `Shape` interface.

```
var shape Shape
shape = Circle{radius: 5.0}
```

Interfaces enable polymorphism in Go, allowing us to work with different types that satisfy a common interface without knowing the exact underlying type.

2. Polymorphism:
Polymorphism is the ability of an object to take on different forms or types. In Go, polymorphism is achieved through interfaces. Since a type can satisfy multiple interfaces, we can use interfaces to define different behaviors for a single type.

Consider the following example:

```
type Sayer interface {
    Say()
}

type Dog struct{}

func (d Dog) Say() {
    fmt.Println("Woof!")
}

type Cat struct{}

func (c Cat) Say() {
    fmt.Println("Meow!")
}

func LetThemSpeak(s Sayer) {
    s.Say()
}
```

In this example, we have two types, `Dog` and `Cat`, both implementing the `Say()` method. The `LetThemSpeak()` function takes a parameter of type `Sayer`, which is an interface. When we pass a `Dog` or `Cat` instance to this function, it calls the `Say()` method specific to that instance. This behavior demonstrates polymorphism, as the function can work with different types that implement the `Sayer` interface.

```
dog := Dog{}
cat := Cat{}

LetThemSpeak(dog) // Output: Woof!
LetThemSpeak(cat) // Output: Meow!
```

Go interfaces define sets of method signatures, and types implicitly satisfy these interfaces. Polymorphism in Go is achieved through interfaces, allowing different types to be treated uniformly based on their shared interface implementation. This flexibility promotes code reusability and abstraction.
https://go.dev/play/p/_wYpgHEzkvU
curriculum item # 136-interfaces-polymorph

Exploring the stringer interface

In the Go programming language, the `Stringer` interface is a predefined interface that **allows types to define their own string representation.** It is part of the `fmt` package and consists of a single method with the signature `String() string`.

Here's the declaration of the `Stringer` interface in Go:

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

To implement the `Stringer` interface, a type simply needs to define the `String()` method with the following signature:

```
func (t T) String() string {
    // Generate and return the string representation of t
}
```

Here, `T` represents the type implementing the `Stringer` interface. Inside the `String()` method, you should generate and return the desired string representation of the object of type `T`.

Once a type implements the `Stringer` interface, it can be used with the `fmt` package to format and print the object using the `Print` and `Printf` functions or in string interpolation. **When a**

value of a `Stringer` type is encountered, the `fmt` package automatically calls the `String()` method to obtain the string representation.

Here's a simple example to illustrate the usage of the `Stringer` interface:

```
package main

import (
	"fmt"
)

type Person struct {
	Name   string
	Age    int
	Gender string
}

func (p Person) String() string {
	return fmt.Sprintf("Name: %s, Age: %d, Gender: %s", p.Name, p.Age, p.Gender)
}

func main() {
	p := Person{Name: "John Doe", Age: 30, Gender: "Male"}
	fmt.Println(p) // Automatically calls the String() method of Person
}
```

In this example, the `Person` struct implements the `Stringer` interface by defining the `String()` method. The `String()` method returns a formatted string representation of the `Person` object. When `fmt.Println(p)` is called, it automatically invokes the `String()` method to obtain the string representation, resulting in the output: "Name: John Doe, Age: 30, Gender: Male".
https://go.dev/play/p/L4TkCiyYn1e
curriculum item # 137-interfaces-polymorph-stringer

Expanding on the stringer interface - wrapper func for logging

- implement a wrapper func that takes type fmt.Stringer
    - log.Println
        - https://pkg.go.dev/log

In the Go programming language, a **wrapper function**, also known as a **wrapper**, is a function that provides an additional layer of abstraction or functionality around an existing function or method. It acts as an **intermediary** between the caller and the wrapped function, allowing you to modify inputs, outputs, or behavior ***without directly modifying the original function.***

Wrapper functions are commonly used for various purposes, such as:

1. **Logging**: A wrapper function can add logging statements before and after invoking the wrapped function. This helps in capturing information about the function calls, input parameters, return values, and any errors that may occur.

2. **Timing and profiling**: Wrappers can be used to measure the execution time of functions, enabling performance analysis and profiling. By recording the start and end times, you can calculate the elapsed time and gather statistics.

3. **Authentication and authorization**: Wrappers can handle authentication and authorization checks before executing the wrapped function. They can validate user credentials, verify permissions, and ensure that the caller has the necessary rights to access the wrapped functionality.

4. **Error handling**: Wrappers can intercept errors returned by the wrapped function and transform them into a different error type or add more contextual information. They can also recover from panics and gracefully handle exceptional situations.

Here's a simple example to illustrate the concept of a wrapper function in Go:

```
package main

import (
	"fmt"
	"time"
)

// Wrapper function for adding timing information
func TimedFunction(fn func()) {
	start := time.Now()
	fn()
	elapsed := time.Since(start)
	fmt.Println("Elapsed time:", elapsed)
}

// Function to be wrapped
func MyFunction() {
	time.Sleep(2 * time.Second) // Simulate some work
	fmt.Println("MyFunction completed")
}

func main() {
	// Call the wrapped function
	TimedFunction(MyFunction)
}
```

In the example above, the `TimedFunction` acts as a wrapper function that measures the elapsed time taken by `MyFunction` to execute. It captures the start time, calls `MyFunction`, calculates the elapsed time, and then prints it. By using the wrapper, you can easily add timing functionality to multiple functions without modifying their implementation.

Please note that the specific implementation of wrapper functions can vary depending on the use case and requirements. Wrappers can be written as higher-order functions that accept function arguments, or they can be implemented as methods of custom types. The choice of implementation depends on the specific needs of your application.
https://go.dev/play/p/oDvKjKxC4A6
curriculum item # 138-interfaces-stringer-wrapper-func

Writer interface & writing to a file

The **writer interface** is a fundamental concept used for **writing data to various output destinations**, including files. The writer interface is defined by the `io.Writer` interface type. It specifies a single method called `Write` with the following signature:

```
func (w Writer) Write(p []byte) (n int, err error)
```

The `Write` method takes a byte slice (`[]byte`) as input and returns the number of bytes written and an error (if any). It writes the contents of the byte slice to the underlying output destination.

To write to a file using the writer interface, you need to create a file or open an existing file using the `os.Create` or `os.OpenFile` function, respectively. These functions return a file that implements the `io.Writer` interface. Here's an example:

```
package main

import (
   "io"
   "os"
)

func main() {
   file, err := os.Create("output.txt")
   if err != nil {
      panic(err)
```

```
    }
    defer file.Close()

    data := []byte("Hello, World!")

    _, err = file.Write(data)
    if err != nil {
        panic(err)
    }
}
```

In this example, we use `os.Create` to create a new file called "output.txt" in the current directory. The `Create` function returns a file (`*os.File`) that satisfies the `io.Writer` interface. We then defer the closing of the file using `file.Close()` to ensure it gets closed properly.

Next, we create a byte slice `data` containing the content we want to write to the file. We call `file.Write(data)` to write the byte slice to the file. The `Write` method returns the number of bytes written and any error encountered during the write operation. If there's an error, we panic and terminate the program.

By utilizing the writer interface, you can write data to files in Go using a consistent and flexible approach. Additionally, this interface is widely used in the Go standard library, enabling compatibility with various output destinations beyond files, such as network connections or other data streams.

THE RELATIONSHIP BETWEEN A **STRING** AND A **[]BYTE**

In Go, a string and a []byte are two different types, but they are closely related and can often be converted between each other.

A **string** in Go represents a sequence of characters. It is an immutable type, which means you cannot modify individual characters within a string. String values are always interpreted as UTF-8 encoded Unicode text.

On the other hand, a **[]byte** is a slice of bytes, where each element represents a single byte. It is a mutable type, so you can modify individual bytes within a byte slice. It can be used to represent binary data or text in various encodings.

Go provides built-in functions to convert between strings and byte slices:

1. Converting a **string to a byte slice:** You can convert a string to a byte slice using the `[]byte()` type conversion. For example:

```
str := "Hello"
```

```
bytes := []byte(str)
```

This will create a new byte slice `bytes` that contains the UTF-8 encoded representation of the string "Hello". Modifying `bytes` will not affect the original string.

2. Converting a **byte slice to a string:** You can convert a byte slice to a string using the `string()` type conversion. For example:

```
bytes := []byte{72, 101, 108, 108, 111}
str := string(bytes)
```

This will create a new string `str` that represents the UTF-8 encoded text corresponding to the byte slice. Modifying `str` will not affect the original byte slice.

It's important to note that converting between strings and byte slices involves encoding and decoding operations, which can introduce potential errors if the encoding is not handled correctly. It's essential to be mindful of the encoding when working with these conversions to ensure accurate representation of the data.

Overall, strings and byte slices provide different ways of representing and manipulating textual or binary data in Go, and Go provides convenient methods to convert between these two types when necessary.

THIS CODE WILL NOT RUN ON THE PLAYGROUND
because a file cannot be created on the playground
but you can look at the code:
https://go.dev/play/p/ysvi-AqkEDA
curriculum item # 139-writer-interface-file

Writer interface & writing to a byte buffer

ABOUT BYTE BUFFER

A byte buffer is a **region of memory** used to **temporarily store a sequence of bytes**. It provides a data structure for efficient manipulation of byte data. A byte buffer allows you to **read and write bytes to and from the buffer,** making it useful for tasks like data serialization, network communication, file I/O, and efficient string manipulation.

The concept of a byte buffer is not specific to any particular programming language but is a general concept in computer programming. Different programming languages may provide their own implementations of byte buffers, each with its own set of features and functionality.

The purpose of a byte buffer is to provide a **flexible and efficient way to work with sequences of bytes**. It typically offers methods or functions for operations such as appending data, reading data, resizing the buffer, and converting the buffer's contents to different types (e.g., strings, integers, etc.).

Byte buffers are commonly used in scenarios where you need to manipulate raw binary data or handle I/O operations that involve byte-level operations. For example, when reading data from a network socket or a file, you can use a byte buffer to efficiently read and process chunks of bytes at a time.

A byte buffer is a data structure that provides a convenient interface to manipulate sequences of bytes efficiently. It serves as a temporary storage for byte data and enables operations such as reading, writing, appending, and resizing byte sequences.

In the Go programming language, `bytes.Buffer` is a type that provides a way to manipulate in-memory byte buffers. It is part of the standard library package called `bytes`.

`bytes.Buffer` is designed to efficiently handle byte manipulation tasks, such as concatenation, appending, reading, and writing to a byte buffer. It provides a convenient interface to work with sequences of bytes, similar to a file or a string.

You can create a new `bytes.Buffer` by using the `NewBuffer` function provided by the `bytes` package:

```
import "bytes"

// Create a new bytes.Buffer
buffer := bytes.NewBuffer(nil)
```

The `nil` argument passed to `NewBuffer` initializes the buffer with an empty byte slice. Alternatively, you can initialize the buffer with an existing byte slice by passing it to `NewBuffer`, like this:

```
data := []byte("Hello, World!")
buffer := bytes.NewBuffer(data)
```

Once you have a `bytes.Buffer`, you can perform various operations on it. Some common operations include:

- Appending data to the buffer using the `Write` or `WriteString` methods.
- Reading data from the buffer using the `Read` or `ReadString` methods.
- Getting the contents of the buffer as a byte slice using the `Bytes` method.
- Getting the contents of the buffer as a string using the `String` method.
- Resetting the buffer to its initial state using the `Reset` method.

Here's an example that demonstrates some basic operations with a `bytes.Buffer`:

~~~
import (
	"bytes"
	"fmt"
)

func main() {
	buffer := bytes.NewBufferString("Hello, ")

	// Append "World!" to the buffer
	buffer.WriteString("World!")

	// Read the contents of the buffer
	data := buffer.String()
	fmt.Println(data) // Output: Hello, World!

	// Reset the buffer
	buffer.Reset()

	// Append new data to the buffer
	buffer.WriteString("Go programming language!")

	// Read the contents of the buffer again
	data = buffer.String()
	fmt.Println(data) // Output: Go programming language!
}
```
~~~

In this example, we create a `bytes.Buffer` initialized with the string "Hello, ". We then append "World!" to the buffer, read the contents, reset the buffer, and append new data. Finally, we read the contents again to see the updated value.

`bytes.Buffer` is a flexible and efficient way to work with byte sequences in Go, often used for tasks like string manipulation, file I/O, or network communication.

https://go.dev/play/p/PqWPMd5X2Om
curriculum item # 140-writer-interface-byte-buffer

Writing to either a file or a byte buffer

In Go, we often use interfaces to allow us to write code that can operate on different types of data. An `io.Writer` is an interface that wraps the basic `Write` method. Both `os.File` and `bytes.Buffer` implement `io.Writer` interface which allows us to write to either a file or a byte buffer in a similar way.

Here's an example of how you can do this:

```go
package main

import (
	"bytes"
	"fmt"
	"io"
	"os"
)

func main() {
	// Writing to a file
	f, err := os.Create("test.txt")
	if err != nil {
		fmt.Println(err)
		return
	}
	defer f.Close()
	writeString(f, "Hello, File!\n")

	// Writing to a Buffer
	var b bytes.Buffer
	writeString(&b, "Hello, Buffer!\n")

	// Printing out the contents of the Buffer
	fmt.Println(b.String())
}

// writeString function uses io.Writer as the type of its parameter.
// Thus, you can pass in any value that satisfies io.Writer - like *os.File or *bytes.Buffer.
func writeString(w io.Writer, s string) {
	_, err := w.Write([]byte(s))
	if err != nil {
		fmt.Println(err)
	}
}
```

In the code above, we define a function `writeString` that takes an `io.Writer` and a string. It then writes the string to the `io.Writer`. Since both `*os.File` and `*bytes.Buffer` implement the `io.Writer` interface, you can use the same `writeString` function to write to a file and a buffer.

The `io.Writer` interface is used extensively in the Go standard library to represent output destinations that you can write bytes to. This makes it very powerful because you can create functions that can write data to various different destinations, as long as they implement the `Write` method.
https://go.dev/play/p/a5-cb_ByMdY
curriculum item # 141-writer-interface-file-or-byte-buffer

Anonymous func

Anonymous self-executing func

- no parameters
- 1 parameter

An anonymous function, often also called a function literal, lambda, or closure, is a function that is defined without being given a name. Anonymous functions can be used in programming where a function is expected but a named function is not necessary or not desirable.

In Go, the concept of anonymous functions is supported. They can be used wherever function types are expected.

Here is an example of an anonymous function in Go:

```go
func() {
    fmt.Println("I'm an anonymous function!")
}()
```

In this example, the function is declared using the `func` keyword, it takes no arguments, and it doesn't return any value. The function's body is enclosed in `{}` and then the function is immediately invoked with `()`.

You can also assign an anonymous function to a variable and then invoke it using the variable name. Here's an example:

```go
printFunction := func() {
    fmt.Println("I'm an anonymous function assigned to a variable!")
}

printFunction() // Call the function using the variable name
```

You can also create an anonymous function that accepts arguments and returns a value. Here is an example:

```go
multiply := func(a int, b int) int {
   return a * b
}

result := multiply(5, 10)
fmt.Println(result) // This will print 50
```

It's important to note that, since Go supports closures, an anonymous function can access and modify variables defined outside of the function itself. This is a powerful feature, but it can also lead to unexpected side effects if not used carefully.

https://go.dev/play/p/p0-fXb-Ugvt
curriculum item # 142-anonymous-func

func expression

Assigning a func to a variable

- **assign a function to a variable**
    - **execute it**
- an expression is something that evaluates to a value
- "first class citizen"

An expression is a combination of ***values, variables, operators, and function calls that are evaluated to produce a single value.*** It can be as simple as a literal value or a variable, or it can involve complex operations and function invocations.

Here are some examples of expressions in Go:

1. Literal values:
- `42`: An integer literal expression representing the value 42.
- `"Hello, World!"`: A string literal expression representing the text "Hello, World!".

2. Variables:
- `x`: A variable expression representing the value stored in the variable `x`.

3. Arithmetic expressions:
- `x + y`: An arithmetic expression that adds the values of variables `x` and `y`.
- `5 * (x - 2)`: An arithmetic expression involving multiplication, subtraction, and parentheses.

4. Function calls:

- `math.Sqrt(16)`: A function call expression invoking the `Sqrt` function from the `math` package with the argument `16`.
- `fmt.Sprintf("Value: %d", x)`: A function call expression invoking the `Sprintf` function from the `fmt` package, formatting the string "Value: %d" with the value of the variable `x`.

5. Logical expressions:

- `x > 10 && y < 20`: A logical expression combining the greater-than and less-than operators with logical AND.

6. Type conversions:

- `float64(x)`: A type conversion expression that converts the value of `x` to a `float64`.

Expressions are the building blocks of Go programs, and they are used to perform computations, manipulate values, and make decisions based on conditions. They can be assigned to variables, passed as arguments to functions, or used in control flow statements like if statements and loops.

The term **"first-class citizen"** refers to the status of certain ***entities, such as values, types, and functions, that are treated equally and have the same capabilities as other entities in the language.*** It means that ***these entities can be assigned to variables, passed as arguments to functions, and returned as values from functions,*** just like any other data type in the language.

In Go, **functions are considered first-class citizens. They can be assigned to variables, passed as arguments to other functions, and returned as values from functions.** This allows Go to support higher-order functions, where functions can operate on other functions. For example, you can define a function that takes another function as an argument and then call that function within the body of the function.

Here's an example that demonstrates the use of first-class functions in Go:

```
package main

import "fmt"

// Function that takes another function as an argument
func applyOperation(a, b int, operation func(int, int) int) int {
	return operation(a, b)
}
```

```
// Functions to be used as arguments
func add(a, b int) int {
	return a + b
}

func subtract(a, b int) int {
	return a - b
}

func main() {
	result1 := applyOperation(5, 3, add)       // Passing 'add' function as argument
	result2 := applyOperation(8, 4, subtract)  // Passing 'subtract' function as argument

	fmt.Println(result1)  // Output: 8
	fmt.Println(result2)  // Output: 4
}
```

In the example, the `applyOperation` function takes two integers and a function as arguments. It applies the given function to the integers and returns the result. The `add` and `subtract` functions are defined separately and passed as arguments to `applyOperation` when calling it.

The ability to treat functions as first-class citizens in Go allows for more flexible and modular code, enabling the use of higher-order functions, function composition, and other functional programming techniques.
https://go.dev/play/p/rd1eQMjlvYD
curriculum item # 143-func-expression

Returning a func

You can return a func from a func. Here is what that looks like.
Returning a function in the Go programming language is a form of using higher-order functions, which are functions that can accept other functions as arguments and/or return other functions as results. This is a common practice in functional programming paradigms, but it's also available in multiparadigm languages like Go.

Here's an example of a function returning another function in Go:

```go
package main

import "fmt"

// This function returns another function
```

```
func incrementer() func() int {
    i := 0
    // Define and return an anonymous function
    return func() int {
        i += 1
        return i
    }
}

func main() {
    increment := incrementer()

    fmt.Println(increment())  // Output: 1
    fmt.Println(increment())  // Output: 2
    fmt.Println(increment())  // Output: 3
}
```

In this code, the `incrementer` function creates a variable `i` and then defines an anonymous function that increments `i` by one each time it's called. This anonymous function captures the `i` variable from its enclosing scope, a feature known as closure in computer science.

The `incrementer` function returns this anonymous function, and then in `main`, we call `incrementer` and assign the returned function to the `increment` variable. This variable is now itself a function that we can call, and each time we call it, it increments its internal `i` counter and returns the new value.

https://go.dev/play/p/mfjfy63Pzii
curriculum item # 144-returning-a-func

Callback

- **passing a func as an argument**

The term **"callback"** in programming refers to **a function or a piece of code that is passed as an argument to another function.** The term itself can be understood by breaking it down into "call" and "back."

The "call" part refers to the action of invoking or executing a function. When a function is called, it starts executing its code, performs certain operations, and then returns a result.

The "back" part refers to the idea that the callback function is called back or invoked by the original function after it has completed its execution or reached a specific point in its code. Instead of the original function returning a result and ending, it "calls back" the callback function, allowing it to execute some additional code or handle specific tasks.

Callbacks are often used in event-driven programming or asynchronous operations, where the flow of execution is not linear. For example, in web development, callbacks are commonly used with JavaScript's asynchronous functions to handle responses from server requests or user interactions.

The term "callback" has become a widely accepted convention in programming to describe this mechanism of passing functions as arguments to other functions and invoking them later. It emphasizes the idea that the callback function is "called back" by the original function, enhancing the understanding of the code's flow and behavior.

```
package main

import "fmt"

// Function that takes another function as an argument
func applyOperation(a, b int, operation func(int, int) int) int {
	return operation(a, b)
}

// Functions to be used as arguments
func add(a, b int) int {
	return a + b
}

func subtract(a, b int) int {
	return a - b
}

func main() {
	result1 := applyOperation(5, 3, add)      // Passing 'add' function as argument
	result2 := applyOperation(8, 4, subtract)  // Passing 'subtract' function as argument

	fmt.Println(result1)  // Output: 8
	fmt.Println(result2)  // Output: 4
}
```

https://go.dev/play/p/7UJCNM0UTWP
https://go.dev/play/p/cm1CRE4o59z
curriculum item # 145-callback-func-as-argument

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

In programming, closure refers to the combination of a function (or a method) and the environment in which it was defined. It allows a function to access variables and bindings that are outside of its own scope, even after the outer function has finished executing.

A closure is created when a nested function references a variable from its containing (parent) function. The nested function retains a reference to the environment in which it was defined, so it can access and manipulate variables from that environment, even if the parent function has already completed.

Closures are particularly useful in situations where you want to create functions that have access to certain variables or data that persist across multiple invocations. They enable you to encapsulate data and behavior together, creating self-contained functions with their own private state.

Closures have various applications in programming, including data privacy, encapsulation, and creating functions with persistent state or behavior. They are widely used in functional programming languages and are a powerful tool for designing elegant and modular code.

https://go.dev/play/p/207g7uGdgUF
curriculum item # 146-closure

Function fundamentals

This is a review of the fundamentals of functions in the Go programming language.
https://go.dev/play/p/S4INA-7i9_v
curriculum item # 147-function-fundamentals

Recursion

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

Recursion in programming refers to the technique of **solving a problem by breaking it down into smaller subproblems of the same type. In other words, a recursive function is a function that calls itself during its execution.**

When a recursive function is called, it solves a smaller instance of the same problem, and then combines the result of the smaller instance with the current instance to obtain the final result. The function continues to call itself on smaller subproblems until it reaches a base case, which is a simple case that can be solved directly without further recursion.

Recursion can be a powerful technique for solving problems that exhibit a recursive structure, such as tree traversal, graph traversal, and many mathematical calculations. However, it's important to design recursive functions carefully, ensuring that they have well-defined base cases and properly handle the termination condition, to avoid infinite recursion.

https://go.dev/play/p/ORGm7AQ3ljU
curriculum item # 148-recursion

Wrapper function

In the Go programming language, a **wrapper function**, also known as a **wrapper**, is a function that provides an additional layer of abstraction or functionality around an existing function or method. It acts as an **intermediary** between the caller and the wrapped function, allowing you to modify inputs, outputs, or behavior ***without directly modifying the original function.*** A wrapper function wraps or modifies another function's behavior.

Wrapper functions are commonly used for various purposes, such as:

1. **Logging**: A wrapper function can add logging statements before and after invoking the wrapped function. This helps in capturing information about the function calls, input parameters, return values, and any errors that may occur.

2. **Timing and profiling**: Wrappers can be used to measure the execution time of functions, enabling performance analysis and profiling. By recording the start and end times, you can calculate the elapsed time and gather statistics.

3. **Authentication and authorization**: Wrappers can handle authentication and authorization checks before executing the wrapped function. They can validate user credentials, verify permissions, and ensure that the caller has the necessary rights to access the wrapped functionality.

4. **Error handling**: Wrappers can intercept errors returned by the wrapped function and transform them into a different error type or add more contextual information. They can also recover from panics and gracefully handle exceptional situations.

In Go programming language, a common practice is to use a wrapper function for error handling. This approach is useful when you want to add more context to the error or handle different types of errors in a specific way.

Here is an example of how you might create a wrapper function to handle errors when reading a file:

```go
package main

import (
    "fmt"
    "io/ioutil"
    "os"
)

func readFile(filename string) ([]byte, error) {
    data, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("readFile %s: %v", filename, err)
    }
    return data, nil
}

func main() {
    data, err := readFile("test.txt")
    if err != nil {
        fmt.Println("Error:", err)
        os.Exit(1)
    }
    fmt.Printf("Data: %s\n", data)
}
```

In this example, the `readFile` function is a wrapper around the `ioutil.ReadFile` function. If `ioutil.ReadFile` returns an error, the `readFile` function wraps this error with additional context (the name of the file that caused the error) and returns this new error.

When the `readFile` function is called from `main`, the error handling logic can use the additional context provided by the `readFile` function to provide more detailed error messages. This approach helps to create more robust and maintainable code by centralizing error handling logic within the wrapper function.

And we same this example earlier of a wrapper function in Go:

```
package main

import (
	"fmt"
	"time"
)

// Wrapper function for adding timing information
func TimedFunction(fn func()) {
	start := time.Now()
	fn()
	elapsed := time.Since(start)
	fmt.Println("Elapsed time:", elapsed)
}

// Function to be wrapped
func MyFunction() {
	time.Sleep(2 * time.Second) // Simulate some work
	fmt.Println("MyFunction completed")
}

func main() {
	// Call the wrapped function
	TimedFunction(MyFunction)
}
```

In the example above, the `TimedFunction` acts as a wrapper function that measures the elapsed time taken by `MyFunction` to execute. It captures the start time, calls `MyFunction`, calculates the elapsed time, and then prints it. By using the wrapper, you can easily add timing functionality to multiple functions without modifying their implementation.

Please note that the specific implementation of wrapper functions can vary depending on the use case and requirements. Wrappers can be written as higher-order functions that accept function arguments, or they can be implemented as methods of custom types. The choice of implementation depends on the specific needs of your application.
https://go.dev/play/p/scleQnXpF8-
curriculum item # 149-wrapper-func

Hands-on exercises

Hands-on exercise #57 - func concepts

- look through all of these concepts & tell yourself about them:
    - functions
    - purpose of functions
        - modular code
        - abstracting code

    - code reusability
      - DRY - Don’t Repeat Yourself
  - parameters vs arguments
  - syntax
    - func, receiver, identifier, params, returns, execute
    - returns
      - multiple returns
      - named returns // Todd doesn't like
  - variadic parameters
  - unfurling a slice
  - defer
  - methods
  - interfaces & polymorphism
    - stringer interface
    - wrapper func
      - logging example
    - writer interface
      - writing to file or byte buffer
  - anonymous func
  - func expressions
    - assigning a func to a variable
  - returning a func
    - functions are first class citizens
  - 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

https://go.dev/play/p/6W8-8CaJjVb
curriculum item # 150-hands-on-57-func-concepts

Hands-on exercise #58 - basic funcs

- 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

https://go.dev/play/p/Jeh_wripOEe

curriculum item # 151-hands-on-58-basic-funcs

Hands-on exercise #59 - variadic func

- 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

https://go.dev/play/p/AVSkvMuaz0H

curriculum item # 152-hands-on-59-variadic-func

Hands-on exercise #60 - defer func

- “defer” multiple functions in main
    - show that a deferred func runs after the func containing it exits.
    - determine the order in which the multiple defer funcs run

https://go.dev/play/p/tbvSX8qy6TT

curriculum item # 153-hands-on-60-defer-func

Hands-on exercise #61 - method

- Create a user defined struct with
    - the identifier “person”
    - the fields:
        - first
        - 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

https://go.dev/play/p/KqOV32q1aC0

curriculum item # 154-hands-on-61-method

Hands-on exercise #62 - interfaces

- create a type **SQUARE**
    - length float64
    - width float64

- create a type **CIRCLE**
    - radius float64
- attach a method to each that calculates **AREA** and returns it
    - circle area= π r 2
        - math.Pi
        - math.Pow
    - 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

https://go.dev/play/p/8BFxl2GXgcw
curriculum item # 155-hands-on-62-interfaces

Hands-on exercise #63 - tests in go #1

- Read the following
- run the code
- read through the code
- try to understand the code

Go provides a built-in testing framework that simplifies the process of testing Go code. Here is an overview of the file structure, naming conventions, and how testing works in Go:

File Structure and Naming Conventions:

1. **Test files:** Test files live in the same package as the code they test. They are named with the following convention: **`filename_test.go`** where filename is the name of the file that contains the code you want to test.

2. **Test functions:** Test functions must start with the word `Test` followed by a word that starts with a capital letter. The signature of a test function is **`func TestXxx(*testing.T)`**, where Xxx does not start with a lowercase letter.

Example:

Suppose you have a function in `main.go` file that adds two integers:

```go
// main.go
package main
```

```
func Add(a int, b int) int {
	return a + b
}
```

To test this function, you would create another file in the same directory called `main_test.go`.

```go
// main_test.go
package main

import "testing"

func TestAdd(t *testing.T) {
    total := Add(5, 5)
    if total != 10 {
        t.Errorf("Sum was incorrect, got: %d, want: %d.", total, 10)
    }
}
```

In the test function, we used the `Add` function and provided it with two integers. We then compared the result with the expected value. If the result is not what we expect, we report an error using `t.Errorf`.

Running the Tests:

To run tests in Go, open your terminal and navigate to the directory containing your files, then type:

```shell
go test
```

If the test fails, the `go test` command will report an error and provide the output from the `t.Errorf` function.

Go's testing package also provides more advanced features like benchmarks, test helpers, and skipping tests. To explore more advanced usage, you can check the official Go testing documentation.

https://go.dev/play/p/GhR7rzvxU3_p

curriculum item # 156-hands-on-63-tests-01

Hands-on exercise #64 - tests in go #2 - unit tests

A **unit test** is a type of software testing where **individual components or units of a**

software are tested. The purpose is to validate that each unit of the software performs as designed. **A unit is the smallest testable part of any software.** It usually has one or a few inputs and usually a single output.

In Go programming language, a unit test usually tests a single function, method, or struct. The goal is to confirm that the behavior is correct.

Unit tests in Go are typically written using the **built-in testing package, `testing`.** This package does not require any third-party libraries, but it's somewhat limited compared to some other languages' testing tools. Despite its simplicity, it has enough features to construct effective unit tests.

When you ask if this differs from a "test" in Go, it's worth clarifying that a "unit test" is a subset of "test". Tests in software can take many forms such as unit tests, **integration** tests, **functional** tests, **system** tests, etc.

An "Integration Test", for example, in contrast to a "Unit Test", would test the interaction between multiple components of the system, to ensure they work together correctly.

So, to summarize, in Go, a unit test is just one kind of test you can conduct, focused on verifying the correct behavior of a small, isolated piece of functionality. Other types of tests have different focuses and may require different tools or techniques.

create a unit test for these three functions

```
package main

import "fmt"

func main() {
	fmt.Printf("%T\n", add)
	fmt.Printf("%T\n", subtract)
	fmt.Printf("%T\n", doMath)

	x := doMath(42, 16, add)
	fmt.Println(x)

	y := doMath(42, 16, subtract)
	fmt.Println(y)
}

func doMath(a int, b int, f func(int, int) int) int {
	return f(a, b)
}
```

```
func add(a int, b int) int {
	return a + b
}

func subtract(a int, b int) int {
	return a - b
}
```

https://go.dev/play/p/ysOWyO9N3Kj

curriculum item # 157-hands-on-64-tests-02-unit-tests

Hands-on exercise #65 - tests in go #3 - unit tests

- create one additional test of any func

https://go.dev/play/p/jxFzGsHyHfQ

curriculum item # 158-hands-on-65-tests-03-unit-tests

Hands-on exercise #66 - documenting code with comments

- **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**
- document our code from the previous exercise

In Go programming, the convention for **documenting code** uses comments immediately preceding the declaration or definition of the item being documented. *This style of documentation is used by the `godoc` tool to automatically generate API documentation.*

1. **Package documentation**: This should be a brief one-liner that begins with "Package `<packagename>`" and provides a high-level overview of the package's purpose. It is typically located in a separate `doc.go` file.

```go
// Package math provides mathematical constants and functions.
package math
```

2. **Function/method documentation:** Comments should be written directly above the function, starting with the function's name.

```go
// Add calculates the sum of two integers.
func Add(x, y int) int {
    return x + y
}
```

3. **Type documentation:** Similar to functions, comments should be written directly above the type definition, starting with the type's name.

```go
// Person represents a person with a name and age.
type Person struct {
    Name string
    Age  int
}
```

4. **Constant and variable documentation:** Follows the same pattern, starting with the constant or variable name.

```go
// Pi is the ratio of the circumference of a circle to its diameter.
const Pi = 3.14159265358979323846
```

Other Go comment conventions include:

- **Use complete sentences.** The first sentence should be a summary starting with the name being declared.
- Prefer to break long lines after punctuation or operators, keeping operands together.
- Do not use URLs in package-level comments. It's better to create a README file for that purpose.

Remember, the goal of comments is to improve code readability and provide context to other developers. It's not about explaining what the code does (the code should speak for itself in that regard), but rather why it does it.

The go doc command shows documentation for exported names (i.e., identifiers that start with a capital letter). Unexported identifiers (i.e., those that start with a lowercase letter) are private to their package and are not included in the output of go doc by default.

However, if you want to see documentation for unexported identifiers in your package while you're developing, you can use go doc -u. This will show the documentation for both exported and unexported identifiers. Note that these options are useful for package developers but are less useful for consumers of the package since they cannot access unexported identifiers.

Please check the current Go documentation or help for go doc (go help doc) for the most up-to-date information, as the behavior and features of Go tools may change.

https://go.dev/play/p/jxFzGsHyHfQ
curriculum item # 159-hands-on-66-documenting-code-with-comments

Hands-on exercise #67 - interfaces & mock testing a db

- Read the following description
- run the code
- read through the code
- try to understand the code

Interfaces in Go are a powerful tool for abstraction and can be especially useful when you want to write **unit tests** for functions that interact with a database. By creating an interface that describes the behavior of the database interactions your code needs, you can swap out the real database for a mock one in your tests.

https://github.com/GoesToEleven/learn-to-code-go-version-03/tree/main/160-hands-on-67-interface-mock-db-test
https://go.dev/play/p/mpDK30B1n5l
curriculum item # 160-hands-on-67-interface-mock-db-test

Hands-on exercise #68 - anonymous func

- Build and use an anonymous func
- anonymous func / func literal / lambda func

an **anonymous** func, aka a function **literal** or a **lambda** function, is a way to define a function without giving it a name. Instead, you can directly declare and define a function inline wherever it is needed. Anonymous funcs are commonly used when you want to pass a function as an argument to another function or when you need to define a short-lived function for a specific task.
https://go.dev/play/p/Ly0L6YEZ-zW
curriculum item # 161-hands-on-68-anon-func

Hands-on exercise #69 - func expression

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

https://go.dev/play/p/fbrC0sX0uAa
curriculum item # 162-hands-on-69-func-expression

Hands-on exercise #70 - func return

- Create a func
    - which returns a func
        - which returns 42
- assign the returned func to a variable
- call the returned func
- print

https://go.dev/play/p/eqpKJ5jCv3g
curriculum item # 163-hands-on-70-func-return

Hands-on exercise #71 - callback

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
    - func square(n int) int
    - printSquare(f func(int)int, int) string

Callback functions are very common in many programming languages including Go. Callbacks are essentially functions that are passed as arguments to other functions and are intended to be called (or "executed") later in your program.

Let's look at a few simple examples.

1. **A Basic Example**:

This first example shows a simple callback function. We're creating a function, `process`, that takes a callback function as an argument. It then uses this callback to process an integer.

```go
package main

import "fmt"

// Our callback function type.
type callbackFunc func(int) int

// A function that takes a callback.
func process(cb callbackFunc, num int) int {
    return cb(num)
}

// Our callback function.
func square(n int) int {
```

~~~
    return n * n
}

func main() {
    result := process(square, 5)
    fmt.Println(result)  // Output: 25
}
```

2. **Callback with Anonymous Function**:

This example uses an anonymous function (or lambda function) as a callback.

```go
package main

import "fmt"

type callbackFunc func(int) int

func process(cb callbackFunc, num int) int {
    return cb(num)
}

func main() {
    // Define an anonymous function that doubles the input.
    doubler := func(n int) int {
        return n * 2
    }

    result := process(doubler, 5)
    fmt.Println(result)  // Output: 10
}
```

3. **Callback in a Iteration**:

The third example iterates over a slice of integers, applying a callback function to each one.

```go
package main

import "fmt"

type callbackFunc func(int) int

func iterate(nums []int, cb callbackFunc) {
~~~

~~~~
    for i, num := range nums {
        nums[i] = cb(num)
    }
}

func main() {
    nums := []int{1, 2, 3, 4, 5}

    // Define a function that triples the input.
    tripler := func(n int) int {
        return n * 3
    }

    iterate(nums, tripler)
    fmt.Println(nums)  // Output: [3 6 9 12 15]
}
```
~~~~

These examples demonstrate how to use callback functions in various contexts in Go, and they should provide a good starting point for understanding this concept.

In Go, `type` is used to define new data types or to create an alias for an existing type.

When we write `type callbackFunc func(int) int`, we're defining a new type, `callbackFunc`, which represents a function that takes an `int` as an argument and also returns an `int`.

This is why, in the `process` function, the `cb` parameter is of type `callbackFunc`. It's saying that `cb` is a function that takes an `int` as an argument and returns an `int`.

So, when we pass the `square` function as the `cb` argument to the `process` function, Go checks to see if `square` matches the `callbackFunc` type. Since `square` is defined as `func square(n int) int` -- a function that takes an `int` as an argument and returns an `int` -- it matches the `callbackFunc` type, and can be passed as the `cb` parameter to `process`.

This is essentially how types work in Go, and how we can define function types that can be used as parameters for other functions. It provides a lot of flexibility and power when structuring our programs, as it allows us to write functions that can take other functions as arguments.

https://go.dev/play/p/o7klLEt0rEE
curriculum item # 164-hands-on-71-callback

Hands-on exercise #72 - closure

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:

https://go.dev/play/p/tEjP1-6S2LW
curriculum item # 165-hands-on-72-closure

Hands-on exercise #73 - wrapper

Here's a simple example to illustrate the concept of a wrapper function in Go:

```
package main

import (
	"fmt"
	"time"
)

// Wrapper function for adding timing information
func TimedFunction(fn func()) {
	start := time.Now()
	fn()
	elapsed := time.Since(start)
	fmt.Println("Elapsed time:", elapsed)
}

// Function to be wrapped
func MyFunction() {
	time.Sleep(2 * time.Second) // Simulate some work
	fmt.Println("MyFunction completed")
}

func main() {
	// Call the wrapped function
	TimedFunction(MyFunction)
}
```

In the example above, the `TimedFunction` acts as a wrapper function that measures the elapsed time taken by `MyFunction` to execute. It captures the start time, calls `MyFunction`, calculates the elapsed time, and then prints it. By using the wrapper, you can easily add timing functionality to multiple functions without modifying their implementation.

Please note that the specific implementation of wrapper functions can vary depending on the use case and requirements. Wrappers can be written as higher-order functions that accept function arguments, or they can be implemented as methods of custom types. The choice of implementation depends on the specific needs of your application.

A wrapper function, also known as a wrapper, is a function that encapsulates or wraps another function or piece of functionality. It is typically used to provide additional behavior or modify the behavior of the wrapped function without directly modifying its code. The wrapper function takes the same arguments and returns the same type as the wrapped function, but it may perform some pre- or post-processing, error handling, logging, or other tasks around the execution of the wrapped function.

Here's an example of a simple wrapper function in Go:

```go
func Logger(f func()) {
    fmt.Println("Starting execution...")
    f()
    fmt.Println("Execution completed.")
}

func main() {
    wrappedFunc := func() {
        fmt.Println("Hello, World!")
    }

    Logger(wrappedFunc)
}
```

In the above example, the `Logger` function is a wrapper that adds logging statements before and after the execution of the wrapped function `wrappedFunc`.

On the other hand, a callback function is a function that is passed as an argument to another function and is invoked by that function at a specific point or event. The purpose of a callback function is to allow the receiving function to execute the callback code when appropriate or necessary. Callback functions are commonly used in event-driven programming or asynchronous operations.

Here's a simple example of using a callback function in Go:

```go
func ProcessData(data []int, callback func(int)) {
```

~~~
    for _, item := range data {
        callback(item)
    }
}

func main() {
    data := []int{1, 2, 3, 4, 5}

    callbackFunc := func(num int) {
        fmt.Println("Processing number:", num)
    }

    ProcessData(data, callbackFunc)
}
```
~~~

In the above example, the `ProcessData` function takes a slice of integers and a callback function. It iterates over the data and invokes the callback function for each item. In the `main` function, we define the callback function as an anonymous function that simply prints the number being processed.

To summarize, a wrapper function wraps or modifies another function's behavior, while a callback function is a function passed as an argument to be executed at a specific point or event.

https://go.dev/play/p/55s4mfHLMdz
curriculum item # 166-hands-on-73-wrapper

Pointers

What are pointers?

All values are stored in memory:

- post office boxes are a good analogy for memory.
    - Every location in memory has an address.
    - **A pointer is a memory address.**
    - "Here's the value *AND* what is the address?"
- lexical elements:

    &

*****

***int**

In the Go programming language, a pointer refers to a variable that holds the memory address. Pointers are a powerful feature that allows you to directly manipulate memory and build complex data structures like linked lists, trees, and more. However, they also require careful use, as improper pointer usage can lead to bugs and errors.

There are two fundamental operations involving pointers:

1. **Address operator** (**&**): The & operator is used to get the memory address of a variable. If \`x\` is an integer variable then \`&x\` will give you a pointer to \`x\` — that is, a memory address where the integer \`x\` is stored.

2. **Dereferencing operator** (*****): The * operator is used to get the value stored at a memory address. If \`p\` is a pointer to an integer then \`\*p\` gives you the integer that \`p\` points to.

Here is an example of pointer usage in Go:

\`\`\`go

~~~
func main() {
    i := 42
    p := &i          // 'p' is a pointer to 'i'
    fmt.Println(*p)   // Dereferencing 'p' gives you the integer that 'p' points to
    *p = 21          // You can change the value that 'p' points to
    fmt.Println(i)    // The value of 'i' is now 21
}
```
~~~

In this example, `p` is a pointer to `i`. You can get the value of `i` by dereferencing `p` with `*p`, and you can change the value of `i` by assigning to `*p`.

Pointers also come into play when you're dealing with functions. If you pass a variable to a function in Go, the function gets a copy of the variable — *everything in go is pass by value* — if the function changes the variable, it won't affect the original. But if you pass a pointer to a variable, the function can dereference the pointer to change the original variable.

Lastly, pointers are crucial when dealing with structs in Go. Since Go doesn’t have classes and objects, you can use structs to create complex types, and you can use pointers to structs to modify these structs or to avoid copying the structs around.

https://go.dev/play/p/rvcrIhHRntg
curriculum item # 167-what-are-pointers

Seeing type & value for pointers

You can print the type and value of a pointer in Go by using the `fmt.Printf` function along with the `%T` and `%v` format verbs. The `%T` verb is used to print the type of a variable, while the `%v` verb is used to print the value in its default format.

For example, if you have a pointer to an integer, you can print its type and value like this:

~~~
```go
package main

import "fmt"

func main() {
    num := 42
    numPtr := &num
    fmt.Printf("Type of numPtr: %T\n", numPtr)
    fmt.Printf("Value of numPtr: %v\n", numPtr)
}
```
~~~

In this code, `numPtr` is a pointer to an integer. The `%T` verb will print the type of `numPtr`, which is `*int`, and the `%v` verb will print the value of `numPtr`, which is the memory address of `num`.
Note: `%v` will print the memory address where the pointer is pointing to. curriculum item #
https://go.dev/play/p/5MqydXuf3pB
curriculum item # 168-printf-pointer-value-type

Dereferencing pointers

Continuing the description from above, if you want to print the value that the pointer is pointing to, you would need to **dereference the pointer using `*`** like this:

```go
fmt.Printf("Value pointed to by numPtr: %v\n", *numPtr)
```

This will print the value of `num`, which is `42`.
https://go.dev/play/p/RAT8EbhVg4J
curriculum item # 169-dereferencing

Pass by value, pointers / reference types, and mutability

In the Go, a **reference type** is pointer – a type of data that refers to, or "points to", the location in memory of a value. Go has several reference types, including **pointers, slices, maps, channels, functions, and interfaces.**

Here's a brief description of each:

1. **Pointers**: A pointer holds the memory address of a value. It allows you to directly access and modify the memory location of a value.

2. **Slices**: A slice is a descriptor of an array segment. It includes a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment).

3. **Maps**: A map is a powerful data structure that associates values of one type (the key) with values of another type (the value). It's an unordered collection of key-value pairs.

4. **Channels**: Channels are used for communication between goroutines (Go's term for threads). They allow you to pass data from one goroutine to another.

5. **Functions**: In Go, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned as values from other functions. When a function is assigned to a variable, the variable stores a reference to the function.

6. **Interfaces**: An interface type represents a set of method signatures. It provides a way to specify the behavior of an object: if something can do this, then it can be used here.

Remember that **when you assign a reference type to another variable, the new variable references the same memory location**.

In Go, all data is **passed by value**, which means that whenever you pass data to a function, Go creates a copy of that data and assigns the copy to a parameter variable. The function can do whatever it wants to the copy without affecting the original data.

Mutability = changeable
However, **mutability** plays a critical role when you consider how "pass by value" works in combination with more **composite / aggregate / complex data** types like slices and maps, and pointers.

A **mutable** value is a value that can be changed. In Go, **slices, maps,** and **pointers** are mutable data types. Even though they are passed by value, they still behave as if they were passed by reference because **the "value" that is copied and passed is the reference to the underlying data**, not the actual data.

For example, let's say you have a slice:

```go
nums := []int{1, 2, 3}
```

And you pass it to a function that modifies it:

```go
func modify(s []int) {
    s[0] = 100
}
```

If you call this function and then print out `nums`:

```go
modify(nums)
fmt.Println(nums)
```

You'll see `[100, 2, 3]`, not `[1, 2, 3]`. This is because the slice header, which includes the pointer to the underlying array, is **passed by value**. But this copied value (the pointer) still points to the same underlying array. So changes made inside the function are visible outside of it, even though the data was passed by value.

In the case of **pointers**, the pointer itself is **passed by value** (meaning the function gets a copy of the address), but the data it points to is the same. Therefore, **dereferencing the pointer and modifying the value it points to inside the function will modify the original value.**

Remember, immutable data types like int, float, string, etc. are safe from this. When passed by value, any changes made in the function won't affect the original value.

In summary, in Go, everything is **passed by value**. However, **for mutable data types, the "value" that's passed includes a reference to the underlying data**, which is why changes made inside the function are visible outside of it.
https://go.dev/play/p/pOn2VmMQ2Fy
curriculum item # 170-pass-by-value-mutability

Pointer & value semantics defined

Value Semantics

Value semantics in Go refers to when the actual data of a variable is passed to a function or assigned to another variable. This means that the new variable or function parameter gets a completely independent **copy of the data.**

Here's a simple example of value semantics:

```go
package main

import "fmt"

func addOne(v int) int {
    v = v + 1
    return v
}

func main() {
    i := 1
    fmt.Println(addOne(i)) // Prints: 2
    fmt.Println(i)      // Prints: 1
}
```

In this example, the function `addOne` doesn't modify the original `i` variable because it's operating on a copy of `i`.

Pointer Semantics

Pointer semantics in Go, on the other hand, involve passing the memory address (a "pointer") rather than the data itself. This means that you can modify the original data, not just a copy of it.

Here's a simple example of pointer semantics:

```go
package main

import "fmt"

func addOne(v *int) {
    *v = *v + 1
}

func main() {
    i := 1
    addOne(&i)
    fmt.Println(i) // Prints: 2
}
```

In this example, the `addOne` function modifies the original `i` variable, because it's operating on a pointer to `i`, not a copy

The choice between **pointer and value semantics** in Go depends on the specific needs of your program.

- **Value semantics** are **simpler** and often **safer** because they avoid side effects, but they can be **inefficient** for large data structures because they involve copying data.
- **Pointer semantics** can be more **efficient** and **flexible**, but they also require careful management to avoid **bugs** related to shared, mutable state.

Value semantics	Pointer semantics
Copied values ("pass by value")	**Shared** memory

https://go.dev/play/p/GhANnabi2kX

curriculum item # 171-pointer-value-semantics-defined

Pointer & value semantics heuristics

There are some general guidelines you can follow when deciding whether to use pointer or value semantics in Go:

1. **Use Value Semantics When Possible**:
   - Value semantics are **simpler** and usually **safer**, since they don't involve shared state or require you to think about memory management.
   - As a rule of thumb, if a function doesn't need to modify its input, or the data you're working with is small (like built-in types or small structs), use value semantics.

2. **Use Pointer Semantics for Large Data:**
   - Copying large structs or arrays can be inefficient.
   - If the data you're working with is large, you might want to use pointer semantics to avoid the cost of copying the data. A rule of thumb: 64 bytes or larger, use pointers.

3. **Use Pointer Semantics for Mutability:**
   - If a function or method needs to modify its receiver or an input parameter, you'll need to use pointer semantics.
   - This is a common use case for methods that need to update the state of a struct.

4. **Consistency:**
   - It's important to be consistent. If some functions on a type use pointer semantics and others use value semantics, it can lead to confusion. Typically, once a type has a method with pointer semantics, all methods on that type should have pointer semantics.

5. **Pointer Semantics When Interfacing With Other Code:**
   - If you're interfacing with other code (like a library or a system call), you might need to use pointer semantics. For example, the `json.Unmarshal` function in the Go standard library requires a pointer to a value to populate it with unmarshalled data.

Remember, these are just guidelines. The specifics can depend on the situation, and sometimes you may have good reasons to make different choices. But these guidelines provide a good starting point.

- VALUE semantics
  - **value semantics facilitate higher levels of integrity**
    - **"the majority of bugs that we get in software have to do with the mutation of memory" ~ Bill Kennedy**
    - **everybody gets their own copy of the data (pass by value) helps keep data bug free**
      - **if everybody is isolated to their own copy of the data (pass by value) then the bug is isolated to that one piece of code**
  - **value semantics also reduce side-effects around concurrency**
  - more likely to **keep values on the stack**

    - you can have methods that use pointers if there's some form of **unmarshaling** going on
- POINTER semantics
    - you don’t want to pass around a lot of data
    - you want to change the data at a location (mutability)

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.

curriculum item # 172-pointer-value-semantics-heuristics

Pointers, values, the stack, & the heap

When you use pointer semantics, it's more likely for values to escape to the heap.

Value semantics & the stack
In Go, when a function receives a value (not a pointer), it gets its own copy of that value. This copy is typically placed on the **stack**, which is fast and doesn't involve any form of garbage collection. Once the function returns, this memory can be instantly reclaimed.

Pointer semantics & the heap
On the other hand, when a function receives a pointer or returns a pointer to a local variable, it indicates to the compiler that this value could be shared across goroutine boundaries or could persist after the function returns. To ensure that the data will remain available, the Go compiler must allocate it on the **heap**, rather than on the stack. ***Heap allocation is more expensive and requires garbage collection.***

Escape analysis
It's important to note that the Go's **escape analysis** decides whether a variable should be allocated on the stack or the heap. Escape analysis examines the function to see if the pointer to a variable is being returned or if the variable is captured within a function literal (closure). If so, the variable escapes to the heap. In other cases, even with pointer semantics, if the variable does not escape, it may be kept on the stack.

To see escape analysis in Go, you use the '-gcflags' flag followed by '-m' when running the `go build` or `go run` command. The '-m' option instructs the compiler to print escape analysis information.

Here's an example:

```bash
```

```
go run -gcflags '-m' main.go
```

You can use `-m=2` for more detailed information:

```bash
go run -gcflags '-m=2' main.go
```

This will print out escape analysis and inlining decisions.

Inlining Decisions

Inlining in programming is an optimization that involves replacing a function call site with the body of the called function. In the **Go compiler**, inlining decisions refer to the choices made by the compiler about when and where to apply this optimization.

Inlining can often make a program run faster due to several reasons:

1. It can save the overhead of a function call (i.e., the time it takes to jump to the function code, set up the function's local variables, and then jump back when the function is complete).

2. It can make further optimizations possible. Once the code from the function is inserted into the calling function, the compiler may be able to see opportunities to optimize this larger block of code that weren't visible when the code was split across multiple functions.

On the other hand, inlining too much code can lead to a larger binary, and possibly, due to cache effects, slower code. Hence, the Go compiler makes inlining decisions to strike a balance, inlining some small and frequently called functions while leaving larger and less frequently called ones alone.

When you run the Go compiler with the '-m' flag, as in `-gcflags='-m'`, it will print information about its inlining decisions along with escape analysis information. You'll see lines like "inlining call to X" where X is a function that the compiler decided to inline.

Do remember, this information is mostly useful for understanding performance characteristics of your Go code, particularly around memory management (specifically, whether certain data stays on the stack or escapes to the heap).

Remember, ***the decision of using pointer or value semantics should be more about sharing and mutating state rather than memory allocation and performance,*** except for performance-critical parts of the code or handling very large data structures.

go run -gcflags -m // shows escape analysis

https://go.dev/play/p/DZJj-OMnFiM

curriculum item # 173-pointers-stack-heap-escape-analysis

Exploring method sets part 1

In Go, a 'method set' is **the set of methods attached to a type. This concept is key to the Go's interface mechanism**, and it is associated with both the value types and pointer types.

- **The method set of a type T consists of all methods with receiver type T.**
    - These methods can be called using variables of type T.
- **The method set of a type *T consists of all methods with receiver *T or T**
    - These methods can be called using variables of type *T.
    - it can call methods of the corresponding non-pointer type as well

The idea of the method set is **integral to how interfaces are implemented** and used in Go. An interface in Go defines a method set, and any type whose method set is a superset of the interface's method set is considered to implement that interface.

A crucial thing to remember is that in Go, if you define a method with a pointer receiver, the method is only in the method set of the pointer type. This is important in the context of **interfaces** because if an interface requires a method that's defined on the pointer (not the value), then you can only use a pointer to that type to satisfy the interface, not a value of the type.

Here's an example to illustrate this:

```go
type T struct {
}

func (t T) M1() {
    // ...
}

func (t *T) M2() {
    // ...
}

type X interface {
    M1()
    M2()
}

func main() {
    var n X
```

~~~~
    t := T{}
    n = t  // This will be a compile error, because T does not implement M2()

    tPtr := &T{}
    n = tPtr  // This is fine, *T implements M1() and M2()
}
```
~~~~

In this example, `T` implements `M1()` and `*T` implements both `M1()` and `M2()`. Therefore, `*T` satisfies interface `X`, but `T` does not. So we can assign `&T{}` to `n`, but not `T{}`.

https://go.dev/play/p/ixftmbd00IV

curriculum item # 174-method-sets-part-1

Exploring method sets part 2

The idea of the method set is **integral to how interfaces are implemented** and used in Go.

- **The method set of a type T consists of all methods with receiver type T.**
  - These methods can be called using variables of type T.

- **The method set of a type *T consists of all methods with receiver *T or T**
  - These methods can be called using variables of type *T.
  - it can call methods of the corresponding non-pointer type as well

https://go.dev/play/p/gI9kgwMULkZ

curriculum item # 175-method-sets-part-2

Hands-on exercises

Hands-on exercise #74 - take an address to create a pointer

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

https://go.dev/play/p/rvcrIhHRntg

curriculum item # 176-hands-on-74

Hands-on exercise #75 - dereference an address

In the provided code, there are variables that store VALUES of TYPE *int and TYPE *string. The values stored are memory addresses. Using the variables provided, do the following:

- print the VALUE stored in each variable
  - these will be memory addresses
- print the TYPE of each variable
- print the data stored at memory locations

    - dereference the stored memory address *

https://go.dev/play/p/7IAjuKBgrNM
curriculum item # 177-hands-on-75

Hands-on exercise #76 - interface implementation & method sets

In the provided code, correct the code to do the following:

- implement the "youngin" interface
    - *you can change anything in the code to do so except the interface*
- maintain consistency within the code
    - receivers should stick with either POINTER or VALUE semantics
- when choosing between POINTER or VALUE semantics, understand why you chose one or the other

https://go.dev/play/p/kG7NIabfkrf
curriculum item # 178-hands-on-76

Hands-on exercise #77 - value & pointer semantics

Create two functions to change a field in a struct called "first" of TYPE string:

- One function will use VALUE SEMANTICS
    - this function will return some VALUE of some TYPE
- The other function will use POINTER SEMANTICS
    - this function will return nothing
- **don't use methods**

HINT:
In Go, a value of type `T` can use a method with a receiver of type `*T` but with some limitations.

If the method has a pointer receiver, `*T`, it means it needs to modify the receiver or the receiver is a large struct that would be expensive to copy.

In this case, if you try to call this method using a value of type `T` (not `*T`), **Go will automatically take the address of that value** and use that pointer as the receiver. However, it is important to note that this is possible only if the value of type `T` **is addressable**, i.e., it is a variable, not an unaddressable value like a literal or a temporary result of a function or expression.

Here is a small example:

~~~
```go
type T struct {
    name string
}
~~~

```
func (t *T) changeName(newName string) {
    t.name = newName
}

func main() {
    var myT T
    myT.changeName("new name") // Works fine, Go automatically takes the address of myT
    fmt.Println(myT.name) // Prints "new name"

    T{"another name"}.changeName("new name") // Compiler error, T{"old name"} is not
addressable
}
```

In the second example, `T{"another name"}.changeName("new name")`, the compiler would throw an error since `T{"another name"}` is not an addressable value. Therefore, a method with a pointer receiver cannot be called on it.

So while a value of type `T` can use a method with a receiver of type `*T`, this only applies to addressable values of type `T`. If the value isn't addressable, you'll get a compile-time error.

FYI, certain things are **addressable** and certain things are not. Generally, you can take the address of variables, elements of an array or slice, fields of an addressable struct, and you can't take the address of constants, literals (like `T{"another name"}`), or the return values of functions or expressions (unless they return a pointer type or an interface type).

The reason that the struct literal `T{"another name"}` is not addressable is because it's a temporary value in memory. It's not stored in a variable or a field, it's just a transient piece of data that's created on the fly and discarded immediately after. It's not allocated a stable location in memory that you can reference, hence it doesn't have an address you can take.

If you want to call a method that has a pointer receiver on a struct, you can create a pointer to the literal directly:

```go
(&T{"another name"}).changeName("new name")
```

In this case, `&T{"another name"}` creates a pointer to the struct literal and you can call methods with a pointer receiver on it.

But remember that any changes made inside `changeName` method will not persist outside the scope of the method call because the original `T` value is a temporary value. This is different from the case when you have an addressable variable where the changes made by the method will persist as they're modifying the actual memory location where the variable is stored.

https://go.dev/play/p/TDnuZTYbHdp

curriculum item # 179-hands-on-77

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/igIGnMv6AN
- **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

Hands-on exercises

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/r8oSG8DIPR**

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/8RKkdtLI6w

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

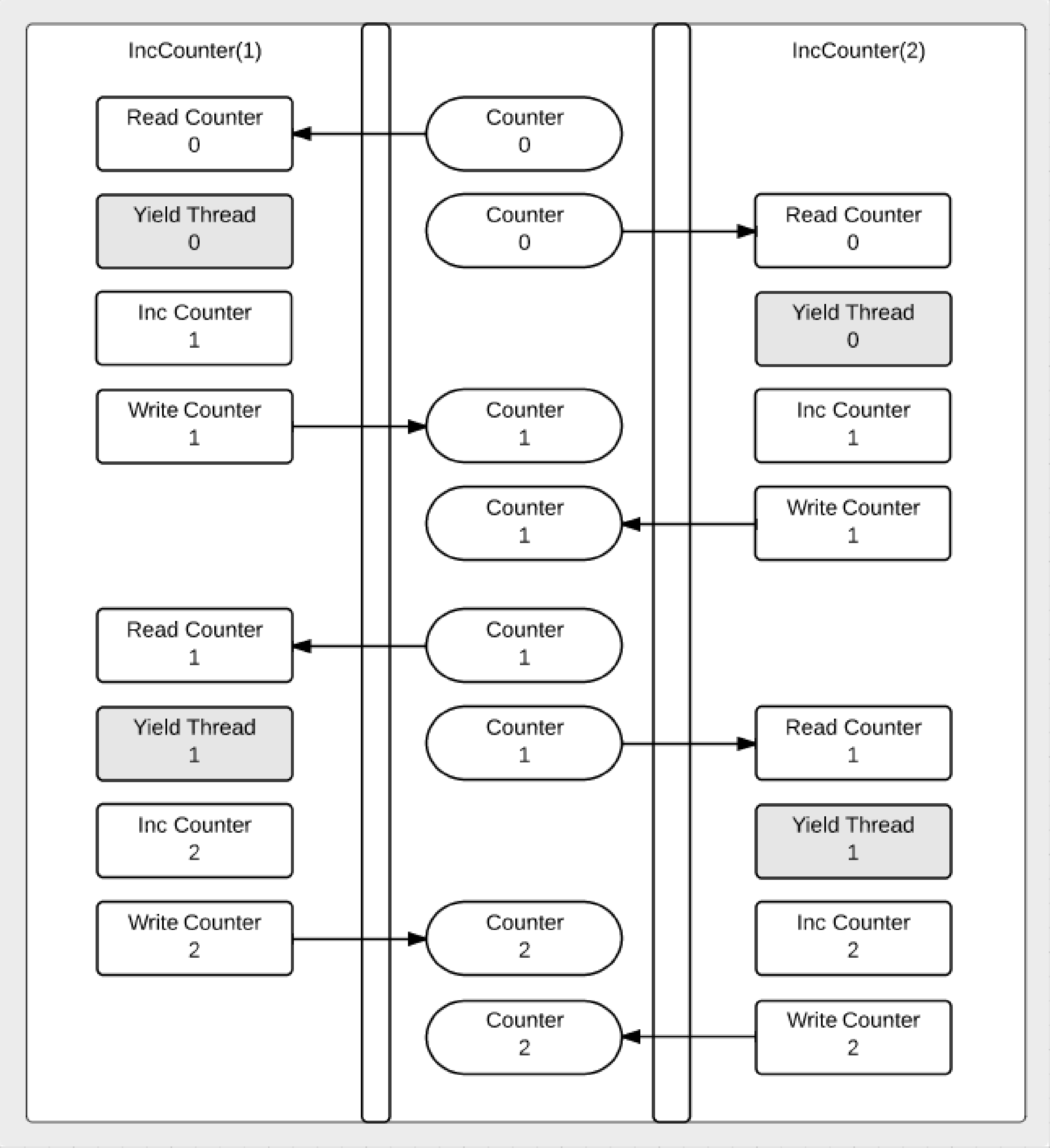
video: 130

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

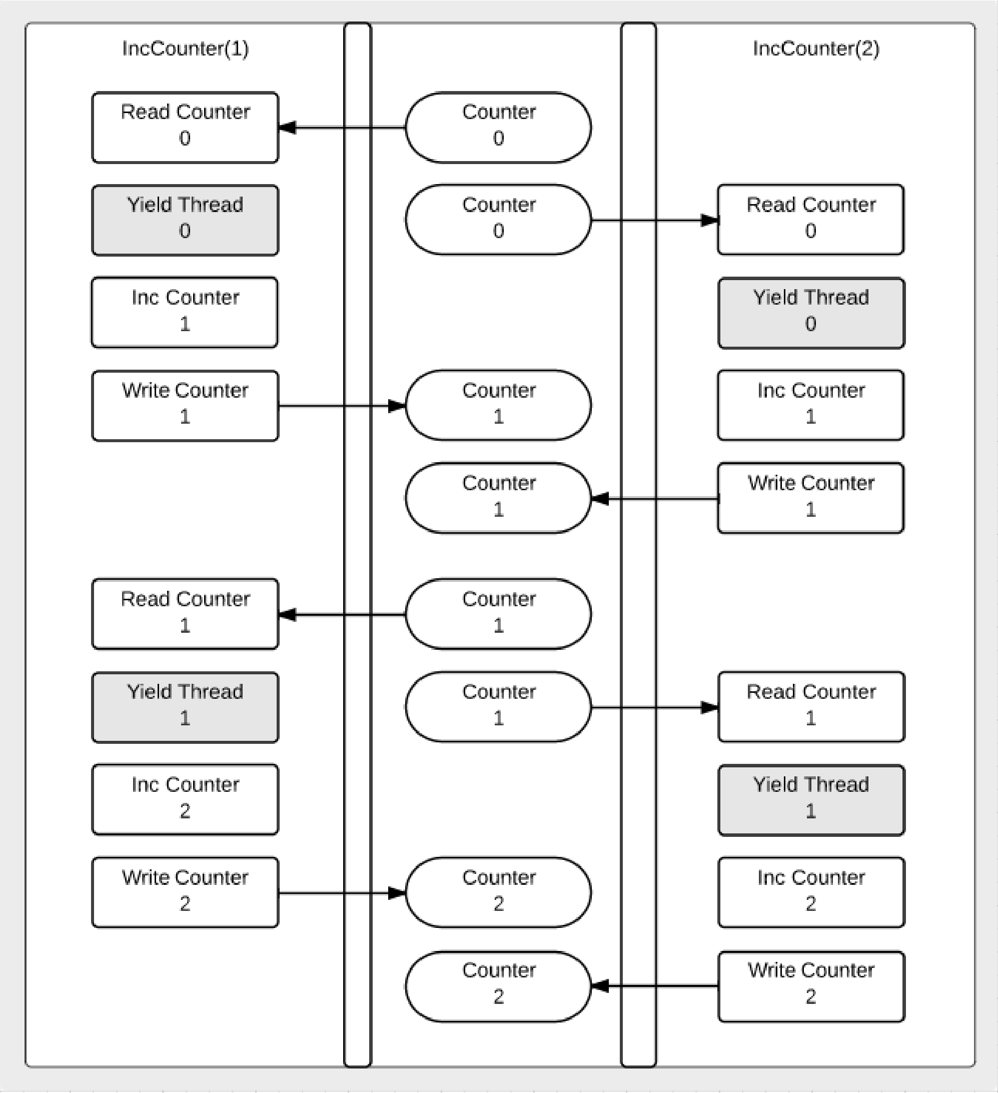
They're called goroutines because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space.
code:

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

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/FYGoflKQej
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

Hands-on exercises

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

Communicating Sequential Processes

Concurrency is a significant aspect of the Go programming language. It provides a way to structure a program that enables it to run more efficiently through the use of multiple processor cores and to deal with many things at once.

Concurrency in Go is achieved mainly through Goroutines and channels, which are primitives provided by the language for dealing with concurrent programming tasks.

Goroutines

A Goroutine is a lightweight thread of execution managed by the Go runtime. It's like a thread in other languages, but it's much cheaper in terms of memory and startup time. You can start a Goroutine by simply using the `go` keyword before a function call:

```go
go funcName() // funcName runs concurrently with the rest of the program
```

Goroutines have the potential to run concurrently, and they're scheduled by the Go runtime, not the operating system, which makes them very efficient. But with Goroutines alone, you can run into problems such as race conditions, where two Goroutines access shared state concurrently.

Channels

This is where channels come in. Channels are a synchronization primitive that allow Goroutines to communicate and synchronize execution. You can send values into a channel and receive those values into another Goroutine. This allows you to coordinate tasks and send data safely between Goroutines:

```go
ch := make(chan int) // Make a channel of type int

go func() {
    ch <- 1 // Send the value 1 into the channel
}()

fmt.Println(<-ch) // Receive the value from the channel and print it
```

Communicating Sequential Processes (CSP)

The concepts of Goroutines and channels are inspired by a formal language and model of computation known as Communicating Sequential Processes (CSP), developed by Sir Tony Hoare. CSP provides a way to describe patterns of interaction in concurrent systems. It's a fundamental part of the design of the Go language.

In CSP, complex systems are built by composing simpler processes that operate independently and interact with each other solely through message-passing communication. This is analogous to Goroutines (processes) communicating via channels (message-passing).

In CSP and in Go, there's no need for locks or condition variables to manage shared state among concurrent threads or processes. Instead, data is sent between Goroutines via channels, and by default, these communications are synchronized: when a value is sent via a channel, the sending Goroutine waits until another Goroutine receives the value.

This approach to concurrency is often summarized by the phrase "Do not communicate by sharing memory; instead, share memory by communicating." This means it's preferred to avoid having multiple Goroutines with access to shared memory and instead use channels to pass the ownership of data between them, preventing race conditions.

By adhering to the principles of CSP, Go provides a rich and expressive framework for writing concurrent and parallel programs that are also safe and easy to reason about.

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/_6bSI5fc17**

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/QWzGUISkJK**
  - **step 3 - change quit from bool to int**
    - **https://play.golang.org/p/tFIvGg9ENT**
- **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

Hands-on exercises

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/Y0Hx6IZc3U

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/FuIKBY5JNj**
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/HI4uG55ait**
- **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

Hands-on exercises

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/Vo5kIoR-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/1ldiBdkdzA**

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.
```

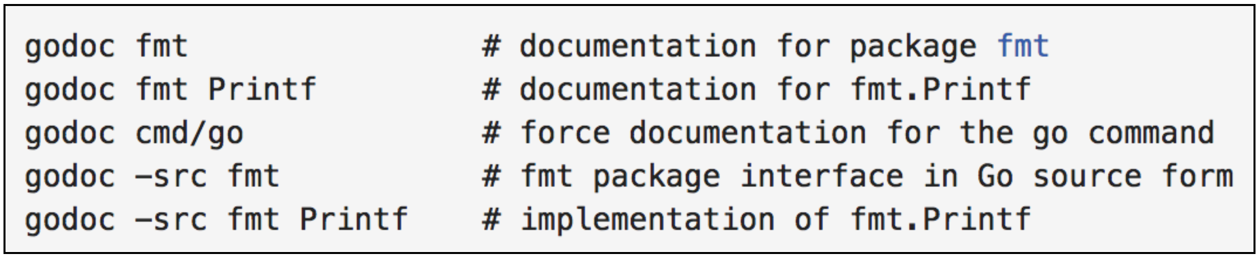
```
At least in the current tree, these invocations all print the
documentation for json.Decoder's Decode method:

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

Hands-on exercises

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

Hands-on exercises

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/P9unTIFeOq**

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

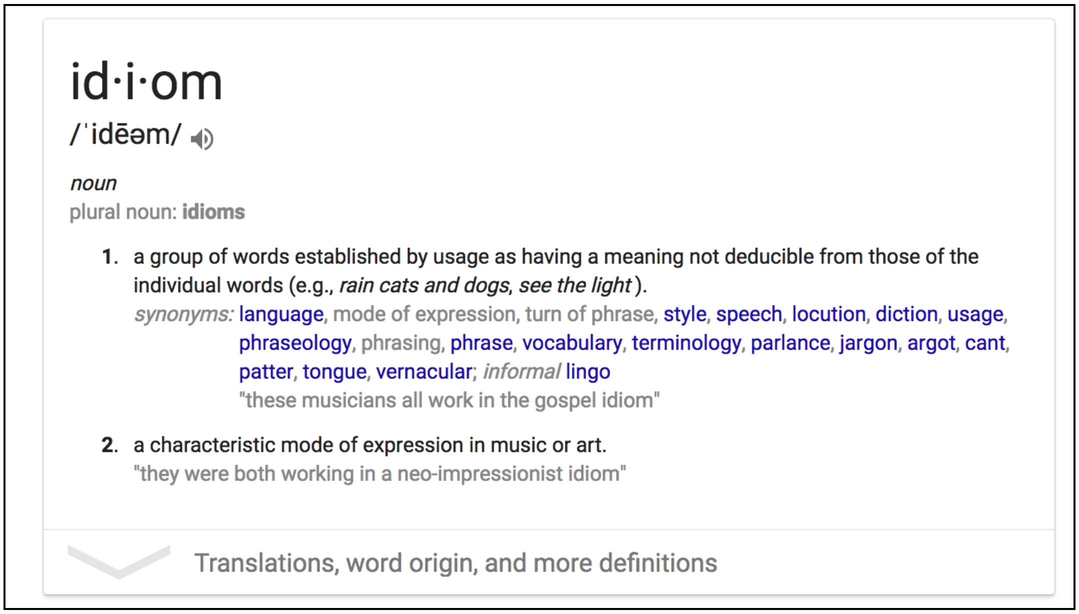
- one folder, many files
  - package declaration in every file
  - package scope
    - something in one file is accessible to another file
  - imports have file scope
- exported / unexported
  - aka, visible / not visible
  - we don’t say (generally speaking): public / private
  - capitalization
    - capitalize: exported, visible outside the package
    - lowercase: unexported, not visible outside the package
- How To Write Packages in Go
  - https://www.digitalocean.com/community/tutorials/how-to-write-packages-in-go
  - "A package is made up of Go files that live in the same directory and have the same package statement at the beginning. You can include additional functionality from packages to make your programs more sophisticated. Some packages are available through the Go Standard Library and are therefore installed with your Go installation. Others can be installed with Go’s go get

command. You can also build your own Go packages by creating Go files in the same directory across which you want to share code by using the necessary package statement."

-

video: 147

- Idiomatic Go code
  - idioms
    - every language has its own language
      - idioms are patterns of speech
    - “idiomatic go”
      - when someone writes “idiomatic Go” they are writing Go code in the way Go code community writes code
    - for example
      - mechanical sympathy
      - pass by value



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

Idiomatic Go code is code that follows the established conventions and best practices of the Go programming language. These conventions and best practices are designed to make the code more readable, maintainable, and efficient.

Some examples of idiomatic Go code include:

Use of camelCase for variable and function names.
Avoiding the use of unnecessary type declarations.
Using the := shorthand for variable declaration and assignment.
Using interfaces instead of concrete types where appropriate.
Avoiding the use of global variables.
Using defer to ensure that resources are released.
Using the panic and recover functions only for exceptional cases.
Using the built-in testing framework to write unit tests.
By following these conventions, Go code becomes more readable and easier to maintain, which makes it more efficient and reduces the likelihood of errors.

Notes - **web**:

- curl -i https://api.github.com/users/GoesToEleven
- https://http.cat/

Notes - **dependency management**:

- our software dependency problem russ cox
    - https://research.swtch.com/deps

Notes - **go commands:**

- **go build -gcflags=-m**
    - escape analysis

In the Go programming language, when you use the `go build -gcflags=-m` command with the `-m` flag, it enables escape analysis and prints optimization decisions made by the compiler. The output provides insights into memory allocation and the movement of objects between the stack and the heap.

When you see the message "escapes to heap," it means that a variable has been allocated on the heap instead of the stack. The **heap** is a region of memory used for dynamic memory allocation, while the **stack** is a region of memory used for function calls and local variables. Variables allocated on the **stack** have a shorter lifetime and are automatically deallocated when they go out of scope, whereas **heap**-allocated variables persist until they are explicitly deallocated.

"Moved to heap" indicates that an object, such as a value or a data structure, has been dynamically allocated on the heap because it needs to outlive the current scope. This often occurs when you assign a reference to a variable declared within a function to a variable declared in an outer scope or when you return a pointer from a function. In such cases, the object is allocated on the heap and the reference to it is moved from the stack to the heap.

The compiler's decision to allocate variables on the stack or the heap is influenced by escape analysis, which determines whether a variable's reference "escapes" from the current scope. If a variable's reference escapes, it means that it is accessed from outside the current scope, and therefore, it needs to be allocated on the heap.

By examining the output of `go build -gcflags=-m`, you can gain insights into how the Go compiler optimizes memory usage and understand whether variables are allocated on the stack or the heap based on their escape behavior.

-

run time polymorphism - empty interface
compile time polymorphism - generics