

Learn How To Code

Google's Go (golang) Programming Language

Course introduction	3
Welcome	3
Questions & answers	4
Getting your certificate	5
Course resources	5
The Go playground	5
Welcome to my house	5
Getting going with Go	5
Why Go?	5
Idiomatic Go code	8
Docs & example code	9
Package main, func main	9
Printing, the fmt package, comments	10
How computers work - core principles	11
ascii, unicode, UTF-8, block comments	11
String literals and documentation	11
Hands-on exercises	11
Hands-on exercise #1	11
The fundamentals of Go	11
Variables, zero values, blank identifier	11
printf, printing verbs, and documentation	12
Using printf for decimal & hexadecimal values	12
Numeral systems: decimal, binary, & hexadecimal	12
Values, types, conversion, housekeeping	13
Values, types, and documentation	15
Hands-on exercises	16
Hands-on exercise #2	16
Hands-on exercise #3	17
Hands-on exercise #4	17
Hands-on exercise #5	18
Hands-on exercise #6	18
Hands-on exercise #7	18
Hands-on exercise #8	18
Programming fundamentals for beginners	19

Introduction	19
Terminology	20
Understanding scope	22
Working at the terminal	22
Using bash on Windows	22
Terminal commands - part 1	22
Terminal commands - part 2	22
Github and ssh authentication	24
Setting up a github repo	24
Checksums	24
Your development environment	25
Getting up and running	25
Running go programs on your machine	25
Go install puts binary in \$GOPATH/bin	26
Go mod and dependency management	26
Introduction to go modules & dependency management	26
Modular code, dependency mgmt, go get - overview	28
Go modules in action: go mod init & go mod tidy	29
Looking at the documentation for go mod tidy	30
Modular code, dependency mgmt, go get - #1	30
Modular code, dependency mgmt, go get - #2	31
Tag git commits with version - overview	32
Tag git commits with version - example #1	34
Tag git commits with version - example #2	34
Specifying dependency version	35
Hands-on exercises	35
Hands-on exercise #9	35
Hands-on exercise #10	35
Hands-on exercise #11	36
Hands-on exercise #12	36
Hands-on exercise #13	36
Hands-on exercise #14	37
Hands-on exercise #15	37
Hands-on exercise #16	37
Hands-on exercise #17 & git clone	37
Housekeeping	38
Hash, encryption, & communication	38
Control Flow	39
Previewing code	39

Understanding control flow	39
If statements & comparison operators	42
Understanding & using Logical operators	42
The "statement; statement" & "comma, ok" idioms	43
Using switch statements to make decisions in code	43
Using select statements for concurrency communication	43
Understanding & using for statement to create loops	44
Multiple iteration - nesting a loop within a loop	45
Understanding & using for range loops	45
Finding a modulus / remainder	45
Hands-on exercises	46
Hands-on exercise #18	46
Hands-on exercise #19	46
Hands-on exercise #20	47
Hands-on exercise #21	47
Hands-on exercise #22	47
Hands-on exercise #23	47
Hands-on exercise #24	47
Hands-on exercise #25	48
Hands-on exercise #26 & infinite loops	48
Hands-on exercise #27	48
Hands-on exercise #28 & a joke	48
Hands-on exercise #29	48
Hands-on exercise #30	48
Hands-on exercise #31	49
Hands-on exercise #32	49
Hands-on exercise #33	49
Hands-on exercise #34	49
Additional code	50

Course introduction

Welcome

- **Getting resources**
 - next to every video
 - "course resources" in this section
 - github
 - link in every video of this section that is not a preview video

- **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

Questions & answers

We are always operating on the edge of understanding.

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

- go lang bridge forum

curriculum item # 002-q-and-a

Getting your certificate

speed me up; autoplay
curriculum item # 003-certificate

Course resources

curriculum item # 004 no video

The Go playground

curriculum item # 005-playground

Welcome to my house

setting expectations
family; dogs; there might be some noise
if it's too much, we'll stop; but the occasionally bark or clunk, we'll keep going!
curriculum item # 006-my-house

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, **GOOGLE**
 - but when google about Go, use golang
- Super cute mascot
 - The **Gopher**
 - <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.

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

/ˈɪdēəm/ 

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.

curriculum item # 008-idiomatic

Docs & 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 # 009-docs-and-example-code

Package main, func main

- 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
 - **golang modules**
 - (these are the same resources as above in previous lecture)
 - [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>
- semicolons are inserted by compiler



<https://go.dev/play/p/xYh5bKHYdh5>

curriculum item # 010-package-main-func-main

Printing, the fmt package, comments

documentation – the truth, written by geniuses

- 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/eDwC_edZ3Ci

curriculum item # 011-fmt-package

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 # 012-how-computers-work

ascii, unicode, UTF-8, block comments

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

curriculum item # 013-ascii-unicode-utf8

String literals and documentation

documentation - know the truth, written by genius

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

curriculum item # 014-string-literals

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 # 015-exercise-01

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 # 016-variables-zero-val-blank-identifier

printf, printing verbs, and documentation

documentation - know the truth, written by genius

- printing a VALUE and TYPE

<https://go.dev/play/p/ePQSTLfIOjz>

curriculum item # 017-printf-verbs-doc

Using printf for decimal & hexadecimal values

documentation - know the truth, written by genius

<https://go.dev/play/p/uvvggBjVF47O>

curriculum item # 018-printf-decimal-hex

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

[understanding numeral systems](#)

curriculum item # 019-numeral-systems

Values, types, conversion, 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, evaluable by the compiler. For instance, $1 < 3$ is a constant expression, while $\text{math.Sin}(\text{math.Pi}/4)$ 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 # 020-val-type-conversion-short-dec

Values, types, and documentation

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 # 021-val-type-documentation

Hands-on exercises

Hands-on exercise #2

Do the "tour of go" through step 17

- <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](https://go.dev/play/p/JXA-lkTdU81)

curriculum item # 022-hands-on-exercise-02

Hands-on exercise #3

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/iZUmqIhqaIC>
 - **(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-03

Hands-on exercise #4

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-04

Hands-on exercise #5

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-05

Hands-on exercise #6

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-06

Hands-on exercise #7

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

- 747
- 911
- 90210

as

- decimal
- binary
- hexadecimal

<https://go.dev/play/p/7cFkjbvbpKuY>

curriculum item # 027-hands-on-exercise-07

Hands-on exercise #8

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-08

Programming fundamentals for beginners

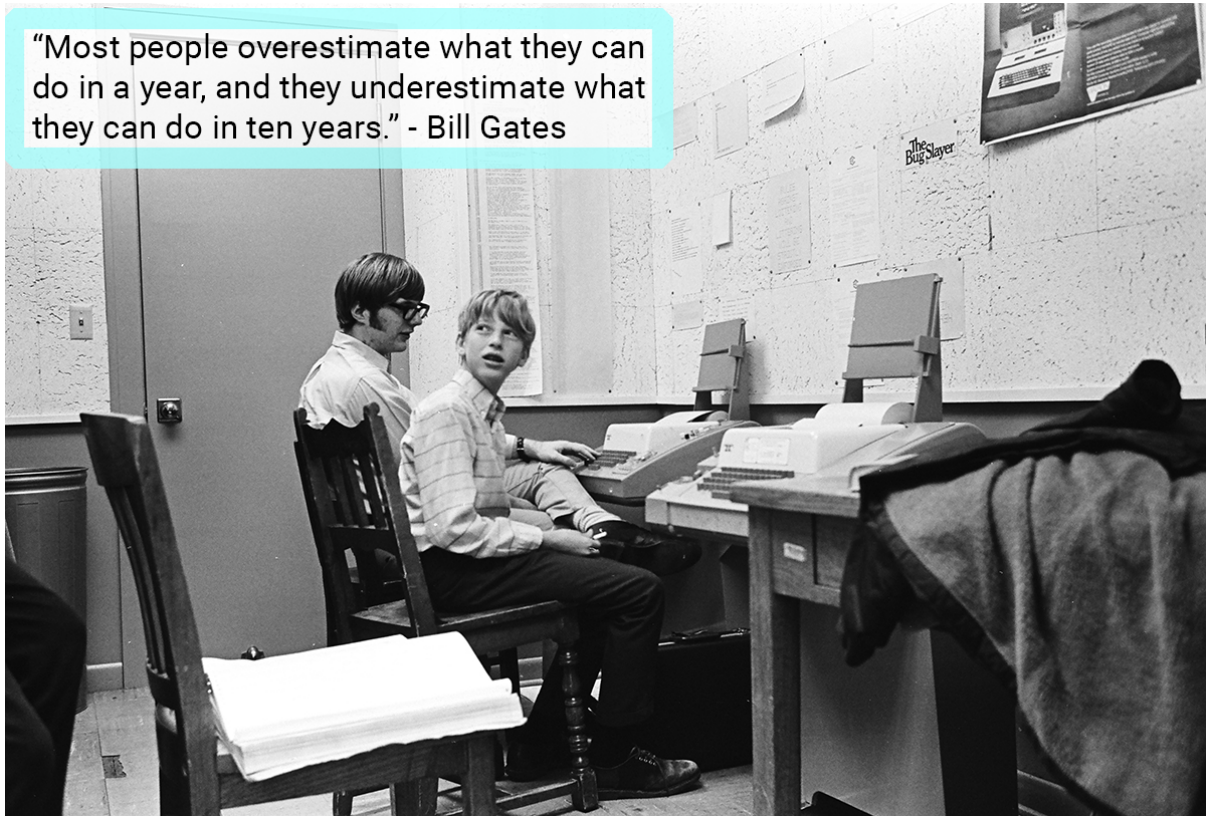
Introduction

- 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

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 are 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

rw-rw-rw-r = 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>

-
- [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/1vp5DlmlMM>
- 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)
 - <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 #9

- 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 #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 #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 #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 #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 #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 #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 #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 #17 & git clone

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>`

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/shchvdllFZe>

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/tlilP-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 #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 #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 #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 #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 #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 #23

- Modify the previous program to use a switch statement

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

Hands-on exercise #24

- there are two parts of 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 #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 #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 #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 #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 #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 occurring

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

Hands-on exercise #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 #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,
    "Money Penny": 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 #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 #33

- use the "statement statement" idiom to
 - initialize x with a 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 #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