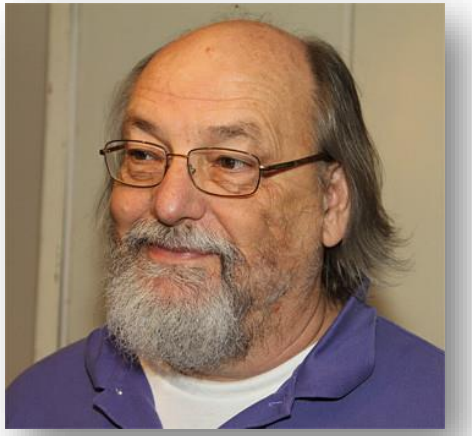
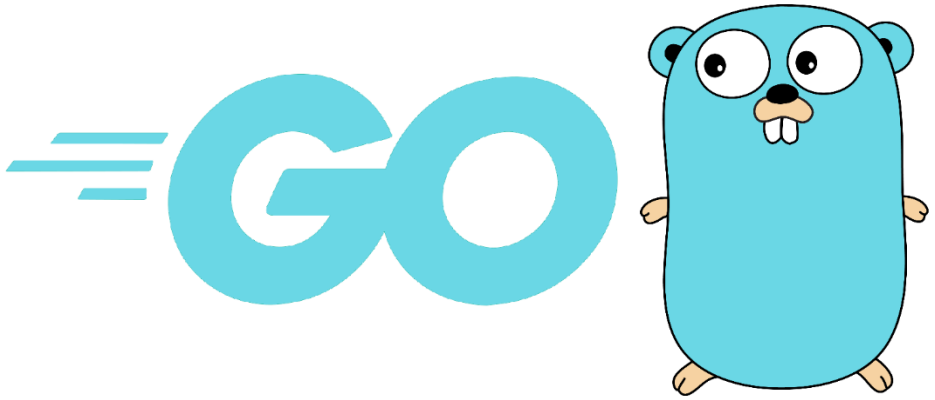
GO BASICS

OVERVIEW

- Written by Google programmers
  - Robert Griesemer
  - Robert Pike
  - Ken Thompson → implemented first Unix OS
    → wrote B programming language (predecessor to C)
- Released in 2012 (first appeared in 2009)
- Compiled, statically typed language
  - Garbage collection
  - Concurrency



MOTIVATION BEHIND GO



- Developers shared dislike of C++
- Easy to create your own package
- Easy to share your packages & use others' packages
- Concurrency → take advantage of multi-core capabilities in modern computers

CURRENT MODERN LANGUAGE OFFERINGS

- C & C++ → fast execution & statically typed
- Python & R → rapid development & easy readability/usability
- Go → bridge the two: fast execution while keeping it simple and developing rapidly

WHY IS GO FAST?

- Only looks at included libraries
  - Java, C & C++ use dependency chain → traverse dependencies of all libraries in chain
- Concurrency → like multithreading but uses less memory
  - Technically just one thread
- Uses Goroutines for Concurrency
  - "main" function is a goroutine
  - Also use ***go*** statement to create a goroutine → go spinner(100 * time.Millisecond)
  - We will use ***net/http*** package to make web requests to turing

GO ON THE TIOBE INDEX

- Monthly rating of programming languages
- TIOBE → The Importance of Being Earnest
  - Oscar Wilde play – 1895
  - Nothing to do with play → company is earnest/sincere in their reporting

Sep 2024	Sep 2023	Change	Programming Language	Ratings	Change
1	1		Python	20.17%	+6.01%
2	3	↑	C++	10.75%	+0.09%
3	4	↑	Java	9.45%	-0.04%
4	2	↓	C	8.89%	-2.38%
5	5		C#	6.08%	-1.22%
6	6		JavaScript	3.92%	+0.62%
7	7		Visual Basic	2.70%	+0.48%
8	12	⇑	Go	2.35%	+1.16%
9	10	↑	SQL	1.94%	+0.50%
10	11	↑	Fortran	1.78%	+0.49%
11	15	⇑	Delphi/Object Pascal	1.77%	+0.75%
12	13	↑	MATLAB	1.47%	+0.28%
13	8	⇓	PHP	1.46%	-0.09%
14	17	↑	Rust	1.32%	+0.35%
15	18	↑	R	1.20%	+0.23%
16	19	↑	Ruby	1.13%	+0.18%

GO ON TURING

- Create a Go directory in your home
  - file permission should be 705 (from CLI, chmod 705 Go).
  - Create 3 subdirectories: ***bin***, ***src***, and ***pkg***
- Edit the .bash_profile file in your home directory to include the following environment variables (the dot in .bash → file is hidden – it's there even if your FTP client doesn't show it):
  - *export GOBIN="$HOME/Go/bin"*
  - *export GOPATH="$HOME/Go/src"*
- Finalize changes on CLI:
  - ***source /etc/profile***
  - ***source ~/.bash_profile***
- Close the shell and re-open and grab mysql driver on CLI:
  - ***go get -u github.com/go-sql-driver/mysql***
  - go env
  - go env -w GO111MODULE=auto

GO REQUIREMENTS

- Go files must end with .go and must be in the src folder.
  - ***package main*** is the exception

Most of our go files will be ***package main***

- To both compile and execute: ***go run filename.go***
- Statements DO NOT terminate with a semi-colon
- When blocking code { } the open curly brace must be on the same line as the control structure → func main ( ) {
- Only for-loops are supported
- &var is a pointer → like a reference variable in Java
- *var is an alias

Example:
m := 2
n := &m
*n = *n * 8 → both m & n are 16

GO REQUIREMENTS con't

- All programs must be associated with a package → again, we will mostly use ***package main***
- All programs begin execution in function main → ***func main ( ) { … }***
- All functions, other than main, begin with an Uppercase letter
  - Includes all functions in external packages (e.g., **math.Abs(-15.3)** **or fmt.Printf("X = %f", x)**)
- All imported packages MUST be used → syntax error if not used
- Use blank identifier, _ , to create blank import
  - Runs init() function but you DO NOT have to explicitly use other methods
  - Example: mysql driver → ***import (***
    _ "github.com/go-sql-driver/mysql"
)

HELLO WORLD

```
package main
import (
  "fmt"
)

func main() {
  fmt.Printf("hello, world\n")
}
```

- Navigate to src folder in CLI
  - ***go run helloWorld.go***

COMMENTS & OPERATORS

- Comments
  - Single line: //
  - Multi-line: /* */
- Operators
  - Unary: ++, --
  - Binary: +, -, *, /, %
  - Compound Assignment: +=, -=, *=, /=, %=
  - Logical: &&, ||, !, and, or, xor
  - String: + (concatenation), += (concatenation append)

VARIABLES

- Begin variables with var → var i, j, k int64
  - NOTE: data type FOLLOWS variable name
  - Most common data types: int, int64, float64, string
- More Examples
  - var ***err*** error
  - var ***name [ ]*** string → slice (auto expand and collapse)
  - var ***p*** Point → pre-defined struct
- Short variable declaration
  - Data type inferred
  - ***kelvin*** := 273.15
- Constants
  - const cm = 2.54
  - const (
    cm = 2.54
    yds = 0.33
)

No commas

TUPLE ASSIGNMENT

- Allows several variables assigned at once
  - a[i], a[j] = a[j], a[i] → swaps i^{th} and j^{th} elements
  - for y != 0 {
    x, y = y, x % y → determines gcd of x & y
    }

STRINGS & MATH

- Must import ***strings*** package
- Lots of commonly used string functions

- Must import ***math*** package
- Lots of commonly used math functions

SELECTION – IF STATEMENT

- if {
  …

  } else {
  …

  }

else is optional, but if there is one it MUST be on same line as }
MAY also have else if
} else if {
…

- if myVal := math.Pow(a, b); myVal < 100 {
  result := myVal
  } else {
  result := 0
  }

Do not use ( ) in condition
Note the semicolon (;) to separate assignment from condition

SELECTION – SWITCH STATEMENT

- No break;

```
switch myVar := x % 3; myVar  {
        case 0:
                …
        case 1:
                …
        case 2:
                …
        default:
                …
}
```

Note the semicolon (;) to separate assignment from expression

- Without an expression, switch operates as an if/else-if
- switch {

```
        case age > 0 && age < 21:  …
        case age < 65:  …
        default:   …
}
```

LOOPING

- NO parentheses in header
- Must block body → { }
- for initialVar := val; condition; increment {

  …

  }
- initialVar and increment are OPTIONAL

- for i:= 0; i < len(mySlice); i++ {

  fmt.Printf("Element: %d", mySlice[i])

  }
- for i, v := range mySlice {

  fmt.Printf("Index: %d, Element: %d", i, v)

  }
- for _, v := range mySlice {

  fmt.Printf("Element: %d", v)

  }

LOOPING con't

- While loops not supported in Go
- Mimic while-loop by not including initialVar and not including increment
- for myVal < 10 {

```
        …
        myVal += 2
}
```

ARRAYS & SLICES

- Slice – describes a piece of an array → size limitation is size of array
- Three ways to define slices
  - a := var [ ] datatype{ value1, value2, etc. }
  - a := make([ ] datatype, length, capacity) → capacity optional; default is capacity = length
  - a := b[low: high] → half-open interval; low included from b, but high is excluded

STRUCTS

- Aggregate data type
- Like a key-value pair
- Similar to associative array
- type Person struct {
  Fname string
  Lname string
  }

Fields begin with uppercase

```
//   hello.go
package main
import "fmt"

func main() {

  // Create a struct
  type Person struct {
   Fname  string
   Lname  string
  }

  me := Person{Fname:"Kristi", Lname:"Davidson"}
  fmt.Println(me.Fname + " " + me.Lname)
  me.Fname = "Gregg"
  fmt.Println(me.Fname + " " + me.Lname)

}
```

STRUCTS con't

```
type Point struct {
  X,Y int
}
type Circle struct {
  Center Point
  Radius int
}
type Wheel struct {
  Tire Circle
  Spokes int
}
```

```
var w Wheel
w.Tire.Center.X = 8
w.Tire.Center.Y = 8
w.Tire.Radius = 5
w.Spokes = 20

w2 := Wheel{Circle{Point{3, 3}, 2}, 15}
w3 := Wheel{
        Tire: Circle{
                Center: Point{X: 3,Y: 3},
                Radius: 2,
            },
        Spokes: 15,
    }
```

Trailing comma REQUIRED