Generate Software Engineering Spring 2026

Bootcamp 2026 Spring

AGENDA

Chiefs intro	1
Expectations	2
Overview of Golang	3
What makes Go special?	4
Threads, coroutines, and goroutines	5
Go’s sync package	6
Go’s channels and channel patterns	7
Go’s context package	8
Dining philosophers made simple	9
Miscellaneous Go tips + follow ups	10

Introductions | Golang





Stone

Tech Chief

4th year + 5th semester in Generate, CS and Mathematics. You can usually find me running, hiking, eating, eating with friends, and of course eating while configuring neovim.

Zach

Tech Chief

3rd year + 4th semester in Generate, CS + Systems, likes to play music, eat other people's food, and doing work from my bed.

Key Objectives

E2E Testing

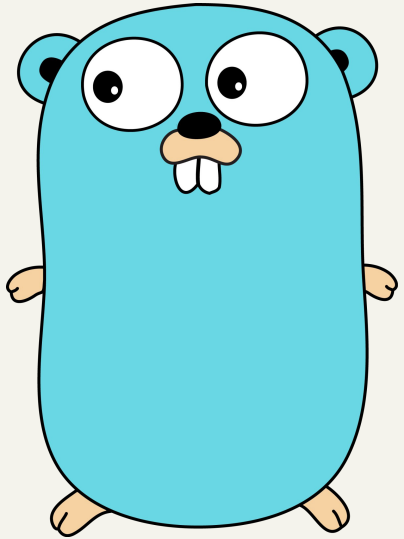
- How to test with DB/HTTP?
- How to write integration/functional tests?
- Do I understand what my code is actually doing?
- Good for your resume.

Deployment

- Deployed software raises morale, allows for user testing, can reveal additional bugs, etc.
- Use tools like Docker and Docker Compose to streamline deployment and development environments.

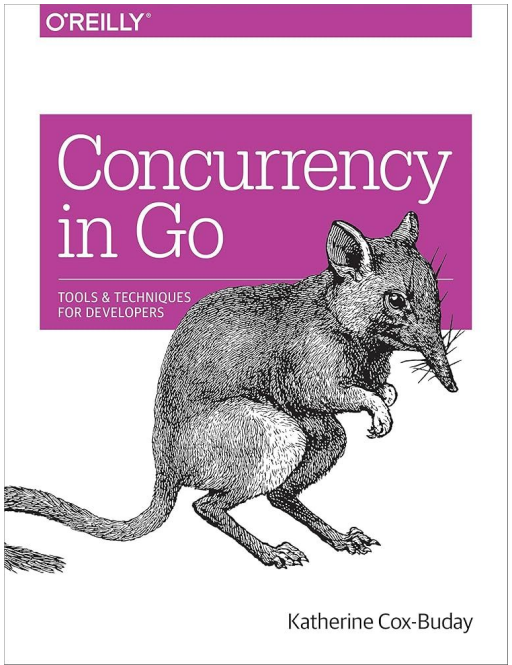
User Testing

- Let's get users!
- Useful for design and engineering.
    - Before building out a triple drop-down search menu, is one search bar fine?



Credits

Exhaustive presentation and code at: https://github.com/zcroft27/team-sync



Overview of Golang: nuts and bolts

- Go is compiled, statically typed, and is exclusively garbage collected.
  - Unless you use the “unsafe” package in Go stdlib, to use malloc and free through invoking C.
- Go is procedural-first, with structs and interface polymorphism.
  - But no implementation inheritance, avoid issues like diamond inheritance in C++/Java/etc.
- Go reverses typical syntax (int x = 3) to (**var x int = 3, or x := 3**).
  - Instead of C’s vague open(...) where you have to search #includes, Go says **os**.Open(...).

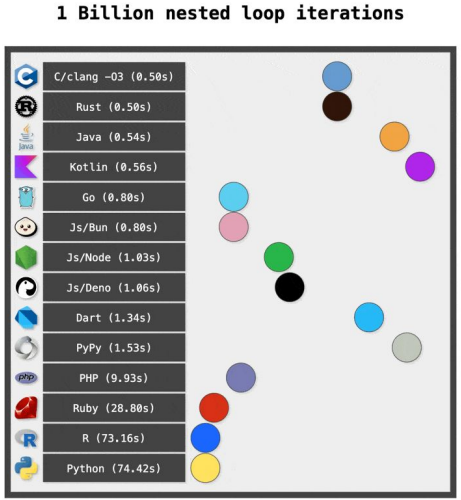
~~- To note in VSCode:~~
  ~~- packages/modules~~
  ~~- Building/executing a binary~~
    ~~- go run vs. go build~~
  ~~- Syntax:~~
    ~~- Variables, pointers, functions, receivers, arrays/slices~~
    ~~- Access modifiers, first class functions, duck typing/interfaces~~

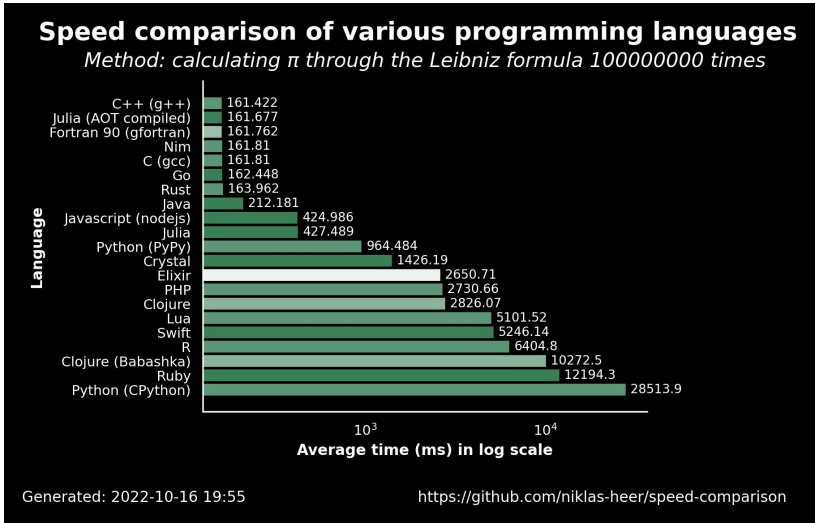
~~... To VSCode~~

Instead, work through: https://go.dev/tour/welcome/

Overview of Golang: speed (cont.)

- While many benchmarks are often iffy because of unfair comparisons or unusual circumstances, Go tends to land slightly slower than Rust/C/C++, but faster than Java, Python, Ruby, JavaScript, etc.





What makes Go special?

- Go has 25 keywords, while Java has 68 keywords. Go is simple by design!
- Go flips the traditional concurrency model from synchronizing memory to communication, following the 1978 ACM paper “Communicating Sequential Processes” (CSP).
  - Formal language for describing patterns of interaction in concurrent systems.
    - Emphasizes that concurrent processes (or functions) should be pure–no synchronized global state, all data is managed through I/O passed between processes.
    - Based on message passing via **channels**, which became popular in distributed systems.

Why should you care?

- CSP-style code, while sometimes long, is easy to reason about and typically safe.
  - * We will come back to this at the end, to handle hungry philosophers.
- CSP directly inspired Go’s **channels** and **select** statement.
- “Don't communicate by sharing memory; share memory by communicating.”

```
PHIL = *[... during ith lifetime ... →
        THINK;
        room!enter( );
        fork(i)!pickup( ); fork((i + 1) mod 5)!pickup( );
        EAT;
        fork(i)!putdown( ); fork((i + 1) mod 5)!putdown( );
        room!exit( )
        ]
```

<- Snippet of (intentionally buggy) CSP code from the paper

Concurrency vs Parallelism

“Concurrency is a property of the code; parallelism is a property of the running program.”

- Parallelism: tasks being run *at the same time*, across multiple cores/processors.
- Concurrency: tasks being structured to run simultaneously, whether they do or not.
  - Concurrent tasks on one core are alternated between rapidly, appearing simultaneous, but are really sequential (time-slicing: preemptive multitasking by CPU, giving each process fixed-length turns).
  - Concurrent code *can* be parallel, but doesn’t have to be.

Threads, coroutines, and goroutines

- OS/kernel threads are 1:1 with hardware cores.
  - 12 cores means 12 tasks can be worked on in parallel.
- Many programming languages support threads. *These are not necessarily 1:1 with kernel threads.*
  - **Native threads** are 1:1 with OS threads. (e.g. POSIX threads in C).
    - Heavy memory consumption, OS manages scheduling, and slow context switches.
  - **Green threads** are M:N (M language threads mapped to N OS threads), and the language runtime manages scheduling between language threads.
- Coroutines:
  - *Concurrency in Go* defines coroutines as: “Concurrent subroutines that are non-preemptive…with multiple points throughout which allow for suspension or reentry.”
  - Similar to green threads: lightweight user-space constructs managed by runtime.
  - Difference lies in scheduling:
    - Green threads are preempted by the runtime.
    - Coroutines are based on cooperative scheduling, voluntarily giving up control at specific yield points.
      - e.g. async/await in JavaScript.

Goroutines

- Goroutines are unique to Go. They are a hybrid between green threads and coroutines.
  - Goroutines don’t define their own suspension or reentry points; Go’s runtime observes the runtime behavior and automatically suspends and resumes when blocked and unblocked.
- Very lightweight (10k empty goroutines ~ 2.8 kb in memory).
  - Native threads would be GBs of memory.
  - Context switch in Go runtime is ~92% faster than OS context switch (benchmark in book).
- There is always at least one goroutine (the main goroutine).

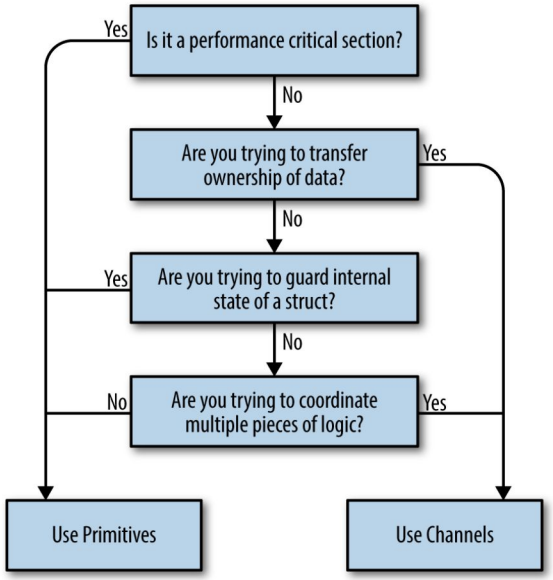
… To VSCode

Memory (GB)	Goroutines (#/100,000)	Order of magnitude
2^0	3.718	3
2^1	7.436	3
2^2	14.873	6
2^3	29.746	6
2^4	59.492	6
2^5	118.983	6
2^6	237.967	6
2^7	475.934	6
2^8	951.867	6
2^9	1903.735	9

Channels

- Derived from CSP, to communicate info between goroutines, *like a stream*.
- For sake of time, skipping details of channels.
- Send values with ‘channelVariable <- value’ operator.
- Read values with ‘value <- channelVariable’ operator.

… To VSCode until patterns



Channels and Channel Patterns

- Channels seem frightening–they can cause blocking, compilation errors, and panics all over.
  - **Assign channel ownership to limit this (confinement).**
- Confinement/Ownership:
- 1. Instantiate the channel.
- 2. Perform writes, or pass ownership to another goroutine.
- 3. Close the channel.
- 4. Encapsulate the previous three things in this list and expose them via a reader channel.

Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

Channels and context package

- Wraps a ‘done’ channel in additional information, for the life of a request.
- Controller -> Model -> Repository layer
- Pass context through each layer, creating sub-contexts for differing behavior.
  - Select on **ctx.Done()** (Done() returns a done channel) in all layers
- Example:
  - Controller: ctx := context.Background() // new, empty context, pass into model
  - Model: ctxWithDeadline := context.WithDeadline(ctx, 2 * time.Second)
    - Pass the context with a deadline to the repository layer.
    - The repository layer selects on the done channel, and will abort after 2 second timeout.

```
func Background() Context
func TODO() Context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
```

Channels and context package (continued)

- Controversial: context.WithValue(parent Context, key any , val any) Context
- An arbitrary grab-bag of **untyped** data (ahhhh scary).

```
func main() {
    ProcessRequest("jane", "abc123")
}

func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), "userID", userID)
    ctx = context.WithValue(ctx, "authToken", authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf(
        "handling response for %v (%v)",
        ctx.Value("userID"),
        ctx.Value("authToken"),
    )
}
```

sync package

- WaitGroup (seen during goroutines)
  - Concurrent-safe counter
    - Calls to *Add(n)* increment the counter by n, and calls to *Done* decrement by 1.
    - Calls to *Wait* block until the counter is zero.
- ErrGroup
  - Immediately cancels other goroutines when any goroutine returns an error.
  - Can set limits on the number of active goroutines.

“Regarding mutexes, the sync package implements them, but we hope Go programming style will encourage people to try higher-level techniques. In particular, consider structuring your program so that only one goroutine at a time is ever responsible for a particular piece of data.”

… To VSCode

Channel Patterns

- Confinement (ownership) pattern
- For-select pattern
- Done channel pattern
  - Read-only channel to signal goroutines to stop their work.
  - Convention is to have ‘<- done’ as the last channel in a select statement.
- The or-channel (the coolest part of this presentation)
  - Multiplex data from n channels onto a single *notification* channel.
    - If you can’t know how many channels you will be reading from at runtime, pass a variadic slice of channels to the or function to return an or-channel.
- Fan-out, fan-in (omitting example code)
  - Multiplex data from n channels onto a single *data* channel, similar to or-channel.
- Tee-channel
  - Like tee command in bash (split input into two outputs):
  - alias testsuite='RAILS_ENV=test bundle exec rake test:parallel | tee testrun.log'

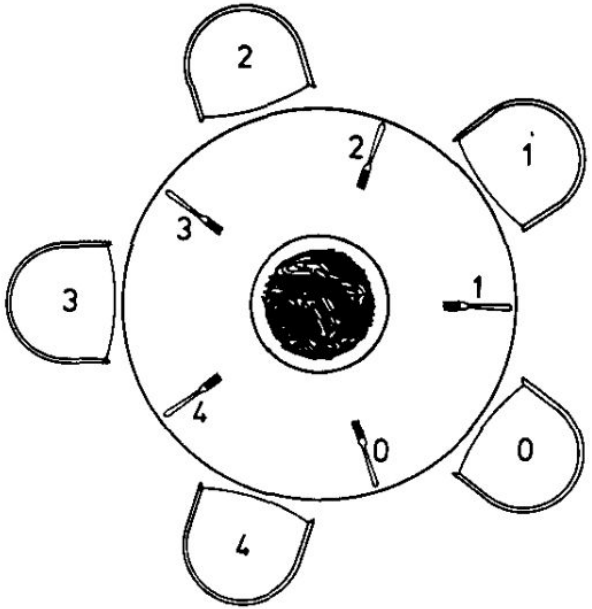
Dining Philosophers: The Return of CSP

- 5 philosophers sit around a table with 5 forks (one between each pair). Each philosopher alternates between thinking and eating.
- Philosophers are very hungry and want two forks to eat!
- To eat, a philosopher needs both adjacent forks (left and right).
- The Challenge: How do they coordinate without:
  - Deadlock - all grab left fork, wait forever for right fork?
  - Starvation - some philosopher never gets to eat?
- Simple representation:
  - Have 5 total forks, and grant 2 at a time to a philosopher.

… To VSCode

Naive implementation with mutexes can easily cause deadlocks.

CSP-style implementation with channels is easy to reason about and is safe!



Pgx CollectRows

- Have you ever wanted to query for multiple rows in Postgres without having to worry about re-writing boilerplate code? Look no further!!

- Follow ups:
  - https://github.com/zcroft27/team-sync further code examples and additional slides
    - More sync package functions
    - ‘synctest’ package new August 2025 for ‘durable bubbles’ and testing concurrent code
  - https://github.com/goccy/go-json very fast drop-in replacement for encoding/json
  - https://go.dev/talks/2012/splash.article Go’s origin story
  - https://antonz.org/go-concurrency/context/ explanation of how to use context
  - https://www.digitalocean.com/community/tutorials/how-to-use-struct-tags-in-go struct tags
    - Directly used in JSON parsing, you will probably see these!