Go Cheatsheet

Tooling & Layout Commands

- Init module: `go mod init example.com/myapp`
- Run/build: `go run .` `go build ./...`
- Test: `go test ./...` Verbose: `-v`
- Format: `gofmt -w .` Imports: `goimports -w .` (external)
- Vet: `go vet ./...`
- Race: `go test -race ./...`
- Benchmark: `go test -bench=. ./pkg`

Project structure (typical)

- `cmd/myapp/main.go` entry point
- `internal/` private packages (recommended)
- `pkg/` public-ish reusable packages (optional)

Basics: Packages, Vars, Types

Hello `package main import "fmt" func main() { fmt.Println("Hello") }`

Declarations

- `var x int = 1`
- Type inference: `var x = 1`
- Short declare (inside funcs): `x := 1`
- Constants: `const Pi = 3.14159`
- Multiple: `var (a int; b string)`

Common types

- `bool, string`
- Integers: `int, int64, uint, uintptr`
- Floats: `float32, float64`
- Byte/rune: `byte` (alias `uint8`), `rune` (alias `int32`)

Zero values

- `0`, `0.0`, `false`, `""`, `nil` (pointers/slices/maps/chans/funcs/interfaces)

Control Flow If

- `if x > 0 {...} else {...}`
- With init: `if v := f(); v > 0 {...}`

For (only loop keyword)

- While-ish: `for cond {...}`
- Classic: `for i := 0; i < n; i++ {...}`
- Infinite: `for {...}`
- Range: `for i, v := range xs {...}`

Switch

- `switch x { case 1: ...; default: ... }`
- Expressionless: `switch { case x>0: ... }`
- Type switch:

`switch v := any.(type) { case string: ...; case int: ... }`

Functions & Multiple Returns Signatures

- `func add(a, b int) int { return a + b }`
- Named return: `func f() (n int) { n=1; return }`

Multiple returns + error

- `v, err := parse(s)`
- Check: `if err != nil { return 0, err }`

Defer

- Runs at function exit: `defer f.Close()`
- Common for unlock/cleanup/metrics

Pointers & Values Pointers

- Address: `&x` Deref: `*p`
- `new(T)` allocates zeroed `*T`

Receiver methods

- Value receiver: `func (t T) M() {...}`
- Pointer receiver (mutates / avoids copying): `func (t *T) M() {...}`

When to use pointer receivers

- Method needs to mutate receiver
- Receiver is large (avoid copies)
- Consistency across methods

Structs, Tags, Constructors Structs

- `type User struct { ID int; Name string }`
- Literal: `u := User{ID:1, Name:"A"}`
- Pointer literal: `u := &User{...}`

Tags (common for json/db) `type U struct { Name string ‘json:"name"‘ }`

Constructor style

- `func NewUser(name string) *User { return &User{Name:nam }`

Arrays, Slices, Maps Arrays (fixed length)

- `var a [3]int`
- `a := [3]int{1,2,3}`

Slices (dynamic view)

- `xs := []int{1,2,3}`
- Length/cap: `len(xs)`, `cap(xs)`
- Append: `xs = append(xs, 4)`
- Subslice: `xs[1:3]`

Copy & delete (slice)

- Copy: `n := copy(dst, src)`
- Delete i:

`xs = append(xs[:i], xs[i+1:]...)`

Maps

- Make: `m := make(map[string]int)`
- Literal: `m := map[string]int{"a":1}`
- Lookup w/ ok: `v, ok := m["k"]`
- Delete: `delete(m, "k")`

Strings & Runes UTF-8 reality

- `string` is bytes (UTF-8 by convention)
- `for i, r := range s {...}` iterates runes

Common ops

- Format: `fmt.Sprintf("%d", n)`
- Convert: `[]byte(s)  string(b)`
- Builder: `var b strings.Builder; b.WriteString("x")`

Interfaces & Errors Interface

- Implicit satisfaction (no `implements`)
- Prefer small interfaces (1–3 methods)

Example `type Reader interface { Read(p []byte) (int, error) }`

Errors

- Sentinel: `var ErrNotFound = errors.New("not found")`
- Wrap: `fmt.Errorf("load: %w", err)`
- Check: `errors.Is(err, ErrNotFound)`
- Typed: `errors.As(err, &target)`

Panic vs error

- Panic for programmer bugs / impossible states
- Errors for expected failures (IO, network, validation)

Generics (Go 1.18+) Basic

- `func Map[T any, U any](in []T, f func(T) U) []U {...}`
- Constraint:

```
type Number interface { int | int64 | float64 }
```

When to use

- Data structures/utilities (sets, queues)
- Avoid for simple one-off code; keep APIs readable

Concurrency Core Goroutines

- Start: `go f()`
- Avoid leaks: have a stop signal or context

Channels

- `ch := make(chan T)` unbuffered
- `ch := make(chan T, n)` buffered
- Send/recv: `ch <- v  v := <-ch`
- Close: `close(ch)` (sender closes)
- Range until closed: `for v := range ch {...}`

Select

- Multiplex:

```
select { case v := <-ch: ...; case <-ctx.Done(): ...; default: ... }
```

Sync primitives

- `sync.Mutex`, `sync.RWMutex`
- `sync.WaitGroup` for joining goroutines
- `sync.Once` init once
- `sync/atomic` for hot counters/flags

Context Pattern (Standard) Pass context everywhere

- `func (s *Svc) Handle(ctx context.Context, ...) error`
- Derive: `ctx, cancel := context.WithTimeout(ctx, d); defer cancel()`
- Check: `select { case <-ctx.Done(): return ctx.Err(); default: }`

HTTP & JSON (Minimal) HTTP server

- `http.HandleFunc("/ping", func(w http.ResponseWriter, r *http.Request){...})`
- `http.ListenAndServe(":8080", nil)`

JSON encode/decode

- Encode: `json.NewEncoder(w).Encode(v)`
- Decode: `json.NewDecoder(r.Body).Decode(&v)`

Time outs (important)

- Set `http.Server{ReadTimeout, WriteTimeout, IdleTimeout}`
- Use `http.Client{Timeout: ...}` or request contexts

Testing & Benchmarks Unit tests

- File: `x_test.go`
- `func TestX(t *testing.T) { t.Fatal(...)}`
- Table-driven:

```
for _, tc := range tests { t.Run(tc.name, func(t *testing.T){...}) }
```

Benchmarks

- `func BenchmarkX(b *testing.B) { for i:=0; i<b.N; i++ {...}}`

Common helpers

- `t.Helper()`
- Subtests: `t.Run`
- Temp dirs: `t.TempDir()`

Common Idioms (Quick Recipes)

Error-first returns `v, err := f(); if err != nil { return err }`

Cancel goroutines

```
ctx, cancel := context.WithCancel(ctx);
defer cancel()
```

Worker pool sketch

- Jobs channel + N workers + WaitGroup
- Close jobs when done producing; workers range over jobs

Functional options (pattern)

- `type Option func(*Cfg)`
- `func WithTimeout(d time.Duration) Option { return func(c *Cfg){...}}`

Avoid nil maps/slices surprises

- Nil slice is OK to `len`/`range`/`append`
- Nil map panics on write; initialize with `make`

Mini Reference

- Exported: identifiers starting with capital letter
- `:=` declares new vars; at least one must be new in scope
- Receiver naming: short (`s`, `r`); be consistent
- Prefer `context.Context` as first param
- Keep goroutine ownership clear (who closes channel?)