Go Basics

Training overview

1. **Go basics**

   Syntax, data structures, interfaces, …

2. **Go basics**

   Best practices, concurrency, benchmarking, profiling, …

3. **Microservices**

   Monoliths, containers, Kubernetes, packaging, docker, …

4. **Microservices**

   CI/CD, skaffold, logging, monitoring, troubleshooting, …

5. **Workshop**

   Building microservices with Go

Part 3

- Review: **no space left on device**
- Goroutines
- Exercise 1
- Channels
- Context
- Data Races
- Exercise 2

Part 4

- Profiling
- Exercise 3
- Error wrapping
- Standard library
- Homework

Review

No space left on device

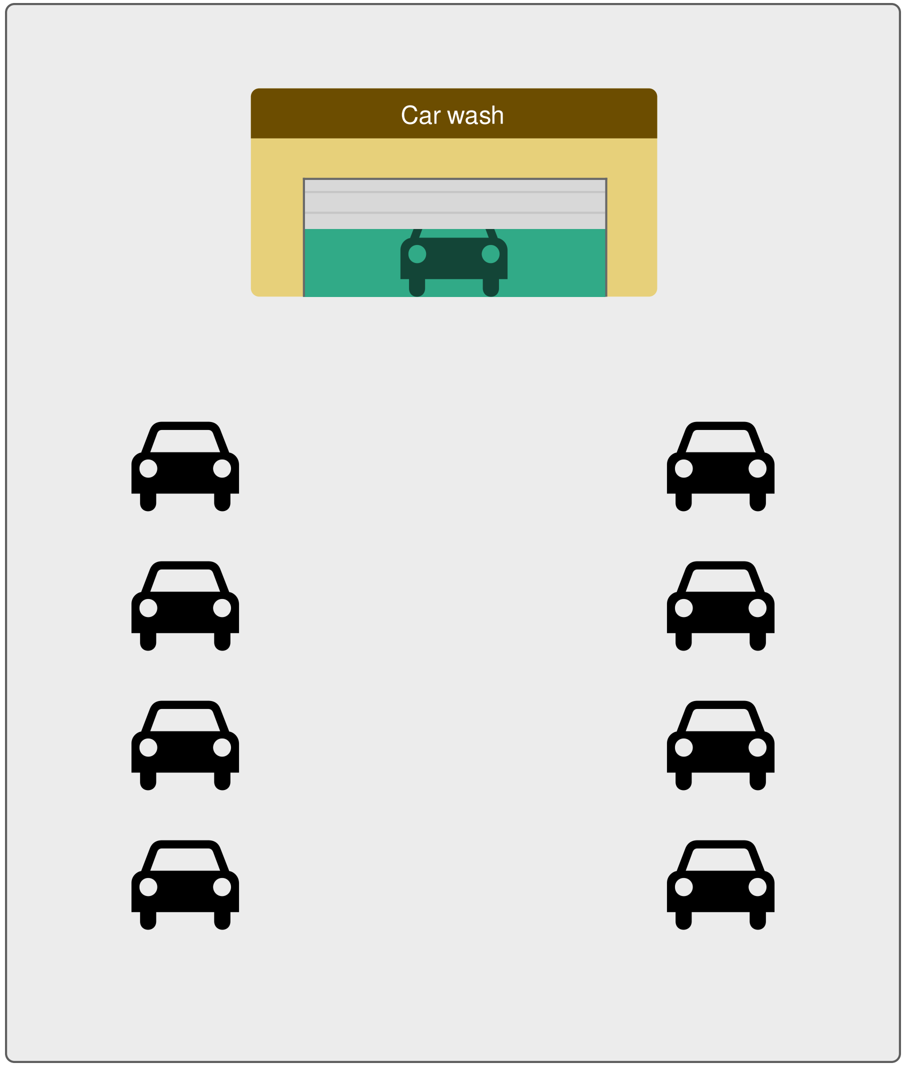
- Who was able to finish the assignment?
- Any questions?
- Demonstration

Goroutines

Concurrency

What?

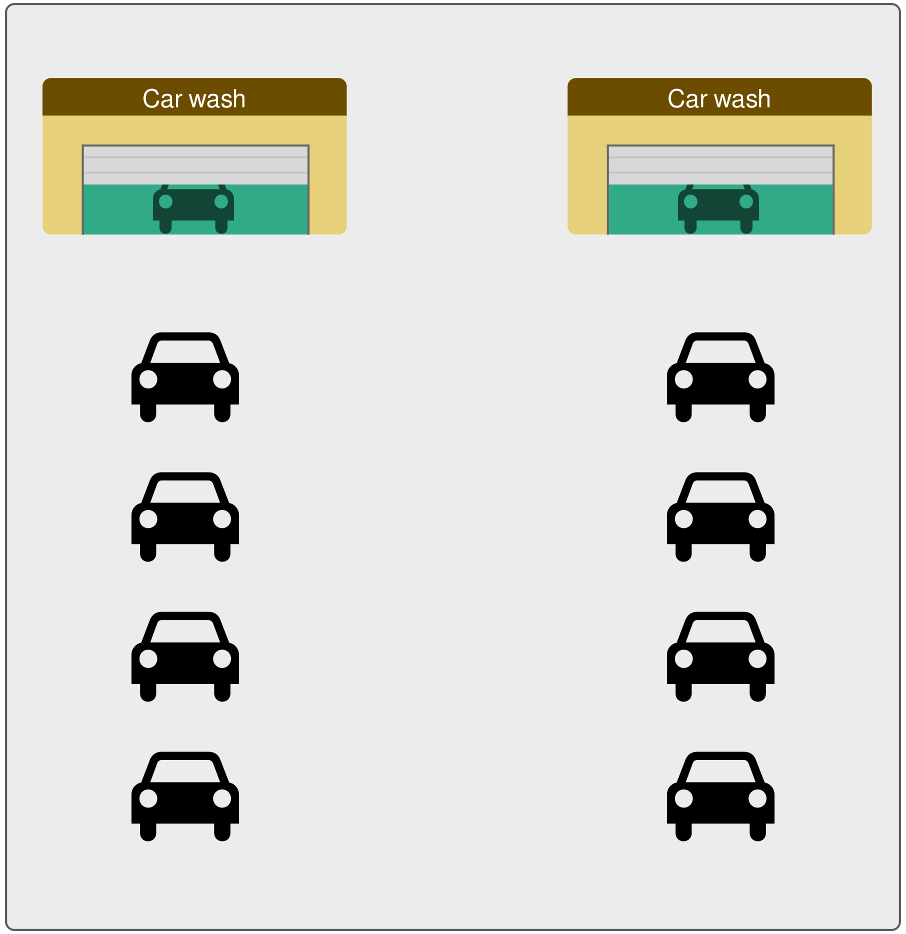
- Composition of independently executing processes
- About structure or technical design



Parallelism

What?

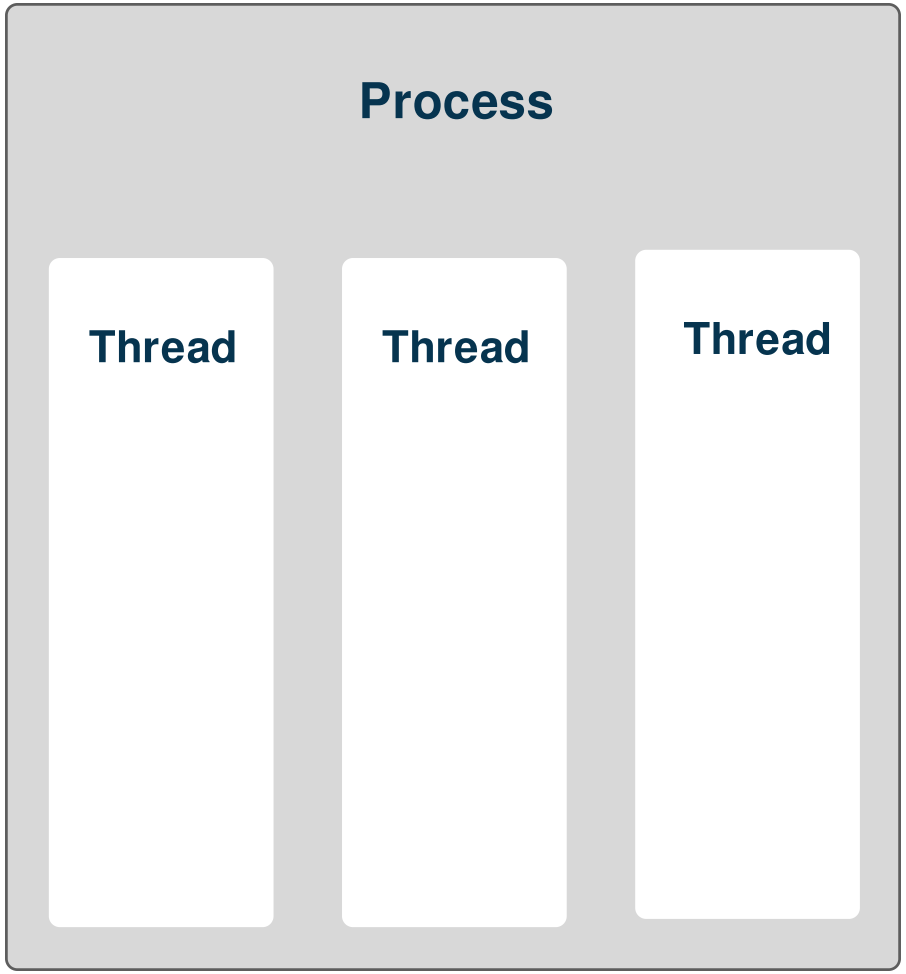
- Simultaneous execution of computations
- About runtime or execution



Threads

Features

- Ordered sequence of instructions
- Can be processed by a single CPU core
- Can be expensive (e.g. context switching)
- Managed by OS



Process

Thread

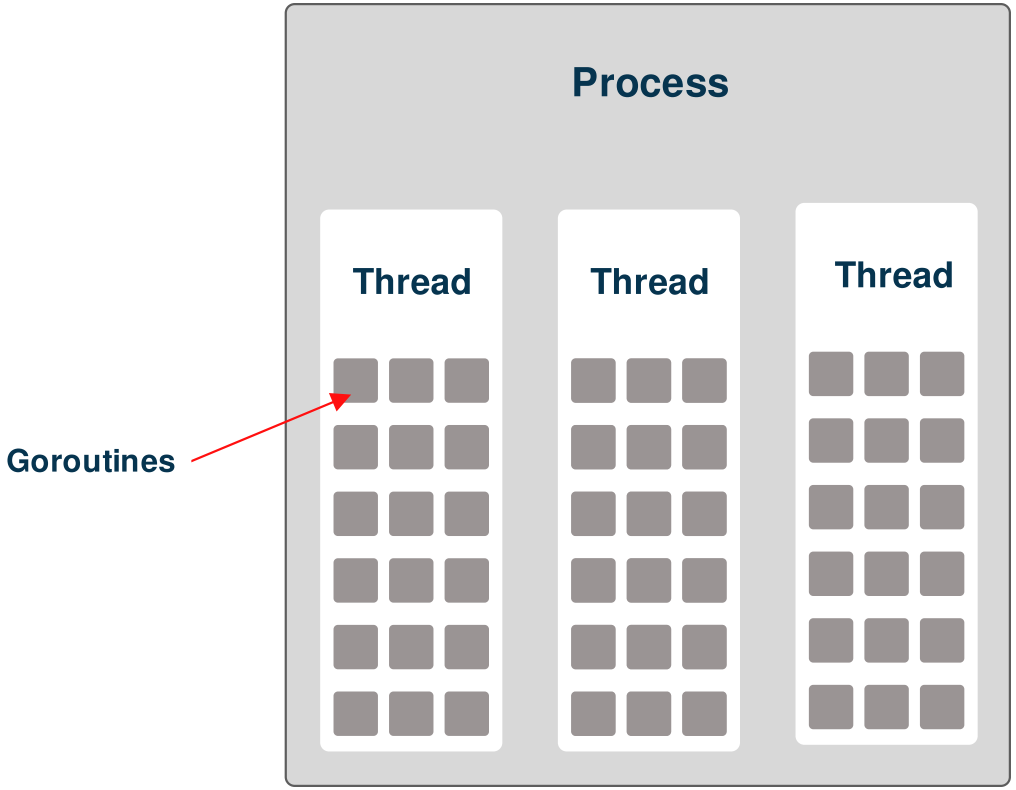
Thread

Thread

Goroutines

Features

- Cheap and lightweight
- `M` goroutines map to `N` threads
- Managed by Go scheduler
- Can be created by using the `go` keyword



Syntax - 1/2

What will the output be?

```go
import "fmt"

func main() {
    go func() {
//  ^^
//  Spawn a new goroutine

        fmt.Println("Who is first")
    }()

    fmt.Println("I wonder")
}
```

Syntax - 2/2

Output

```
I wonder
```

Why?

- Program is started
- `go` keyword spawns a new goroutine
- `"I wonder"` is printed on the screen
- Program exits before goroutine is executed

Waiting - 1/2

Summary:

- Use `WaitGroup` to wait for goroutines to finish
- `.Add(n int)` increments the counter by `n`
- `.Done()` decrements the waitgroup counter
- `.Wait()` blocks until the counter is zero

Waiting - 2/2

```go
func main() {
    var wg sync.WaitGroup

    wg.Add(1)

    go func() {
        fmt.Println("Who is first")
        wg.Done()
    }()

    fmt.Println("I wonder")
    wg.Wait()
}
```

What if `wg.Done()` is never called?

- Program will hang forever, error prone
- Use `defer wg.Done()` instead

Example

```go
var wg sync.WaitGroup

go func() {
    defer wg.Done()
//  ^^^^^
//  Will be deferred until function exits

    fmt.Println("Who is first")
}()
```

Exercise 1

Summary

- We want to perform 100 requests to a server
- Each request is computationally expensive
- We want to do this concurrently

Go to

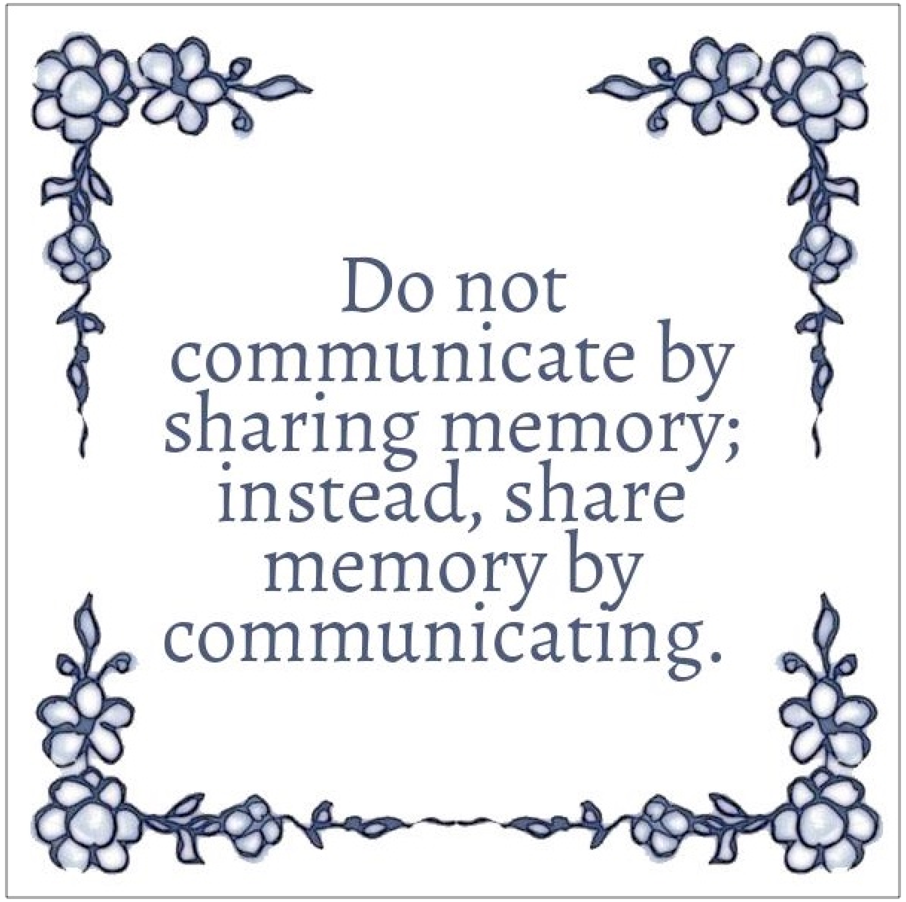
training.brainhive.nl

Review

Channels

Concurrent programming

- Traditional languages require sharing memory
- Use channels to communicate between goroutines
- If nothing is shared, no data races can occur



Do not communicate by sharing memory; instead, share memory by communicating.

Syntax

```go
ch := make(chan int)
//    ^^^^ ^^^^ ^^^
//    Create channel of type `int`

go func() {
    ch <- 100
    // ^^
    // Send a value to the channel
}()

value := <-ch
//       ^^^^
//       Read from channel

fmt.Println(value)
// Output: 100
```

Breakdown

Create a channel

```go
ch := make(chan int)
```

Send a value to the channel

```go
ch <- 150
```

Read from channel

```go
value := <-ch
// Output: 150
```

Unbuffered channels

Summary

- Unbuffered when buffer size: `= 0`
- Sending and receiving is blocking

Example

```
make(chan int)
make(chan int, 0)
```

Buffered channels

Summary

- Unbuffered when buffer size: `> 0`
- Sending is blocking when buffer is full
- Receiving is blocking when buffer is empty

Example

```
make(chan int, 10)
```

For range

Summary

- Can be used to continuously read from channel
- Will be blocking until the channel is closed

Example

```go
ch := make(chan int, 1)
ch <- 10
close(ch)

for i := range ch {
    fmt.Println(i)
    // Output: 10
}
```

Closing channels - 1/2

Summary

- Use `close(ch)` to close a channel
- Sending on a closed channel will cause a panic

Example

```go
ch := make(chan int, 1)
ch <- 10

close(ch)
```

Closing channels - 2/2

In general:

- Only the sender should close a channel, never the receiver
- Channels aren't files, closing them is most likely not required
- There are some exceptions (to terminate a `range` loop)

Select statement - 1/2

Summary

- Can be used to wait on multiple operations
- Blocks until one of the operations finishes
- If multiple operations are ready, one is chosen at random

Example

```
select {
    case value1 := <-ch1:
        fmt.Println(value1)
    case value2 := <-ch2:
        fmt.Println(value2)
}
```

Select statement - 2/2

Default case can be used to avoid blocking

Example

```go
ch := make(chan int)

select {
    case value := <-ch:
        fmt.Println(value)
    default:
        fmt.Println("No value received")
}
```

Context - 1/3

Does two things:

1. Cancellation
2. Passing request-scoped values

Summary

- Carries deadlines, cancellation signals, and other request-scoped values
- Most Go packages accept a `context.Context` as the first argument
- Can be used to cancel long running operations or implement timeouts

Context - 2/3

Example

```go
ctx := context.Background()

ctx, cancel := context.WithTimeout(ctx, time.Second * 30)
defer cancel()

doSomething(ctx)
// Needs to handle context properly
```

Context - 3/3

Support for cancellations

```go
ctx := context.Background()

ctx, cancel := context.WithTimeout(ctx, time.Second * 30)
defer cancel()

ch := make(chan int)

select {
    case <-ctx.Done():
        fmt.Println("Context was cancelled")
    case value := <-ch:
        fmt.Println(value)
}
```

Data races

What will `a` / `b` be?

```go
a := 1
b := 2

go func() {
    a *= 2
    b *= 3
}()

a += 5
b += 10

fmt.Println(a)
fmt.Println(b)
```

Result

- Value will be `6 / 12` or `6 / 36`
- Depends on which path is faster
- Result is not deterministic
- This is called a "data race"
- We can detect this

Race detector

- Go includes a race detector
- Can be enable by using `-race` and works with `go test` / `go run`

How?

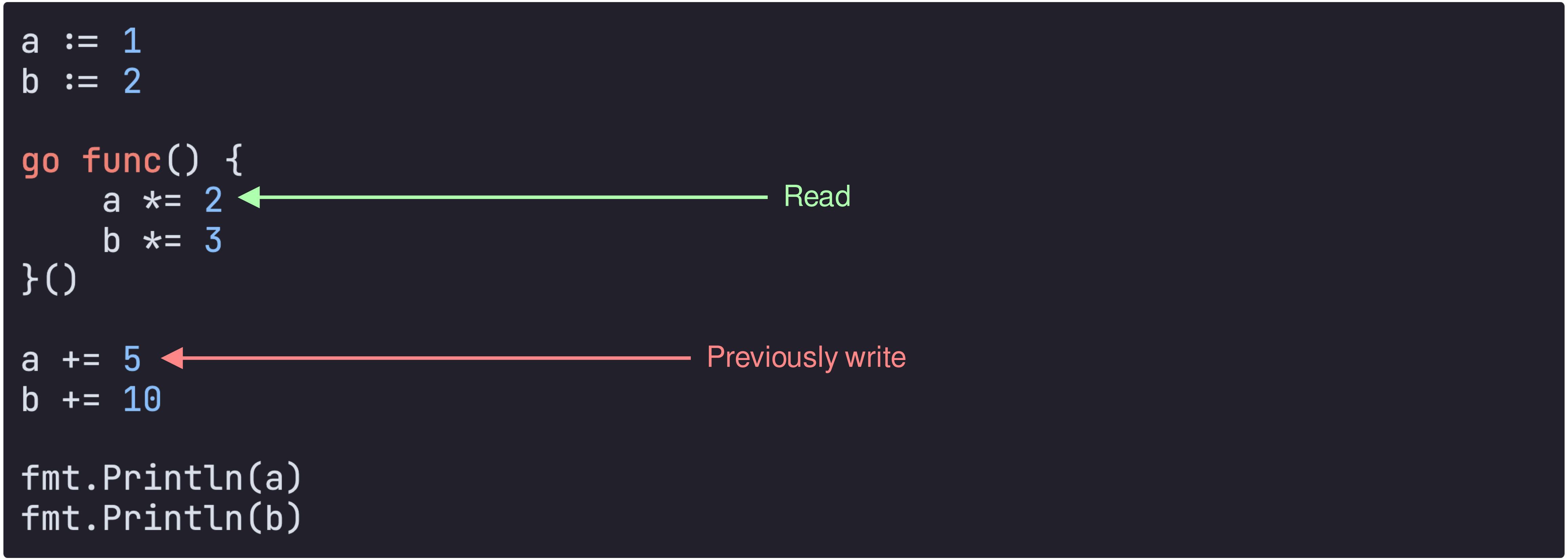
- Runs the program with the race detector enabled

Example

```
$ go run -race source.go
```

```
==================
WARNING: DATA RACE
Read at 0x00c0000b4018 by goroutine 7:
  main.main.func1()
      /code/race.go:10 +0x34

Previous write at 0x00c0000b4018 by main goroutine:
  main.main()
      /code/race.go:14 +0x118

Goroutine 7 (running) created at:
  main.main()
      /code/race.go:9 +0x100
==================
==================
WARNING: DATA RACE
Read at 0x00c0000b4028 by goroutine 7:
  main.main.func1()
      /code/race.go:11 +0x5c

Previous write at 0x00c0000b4028 by main goroutine:
  main.main()
      /code/race.go:15 +0x140

Goroutine 7 (running) created at:
  main.main()
      /code/race.go:9 +0x100
==================
Found 2 data race(s)
exit status 66
```

Visual representation - 1/2

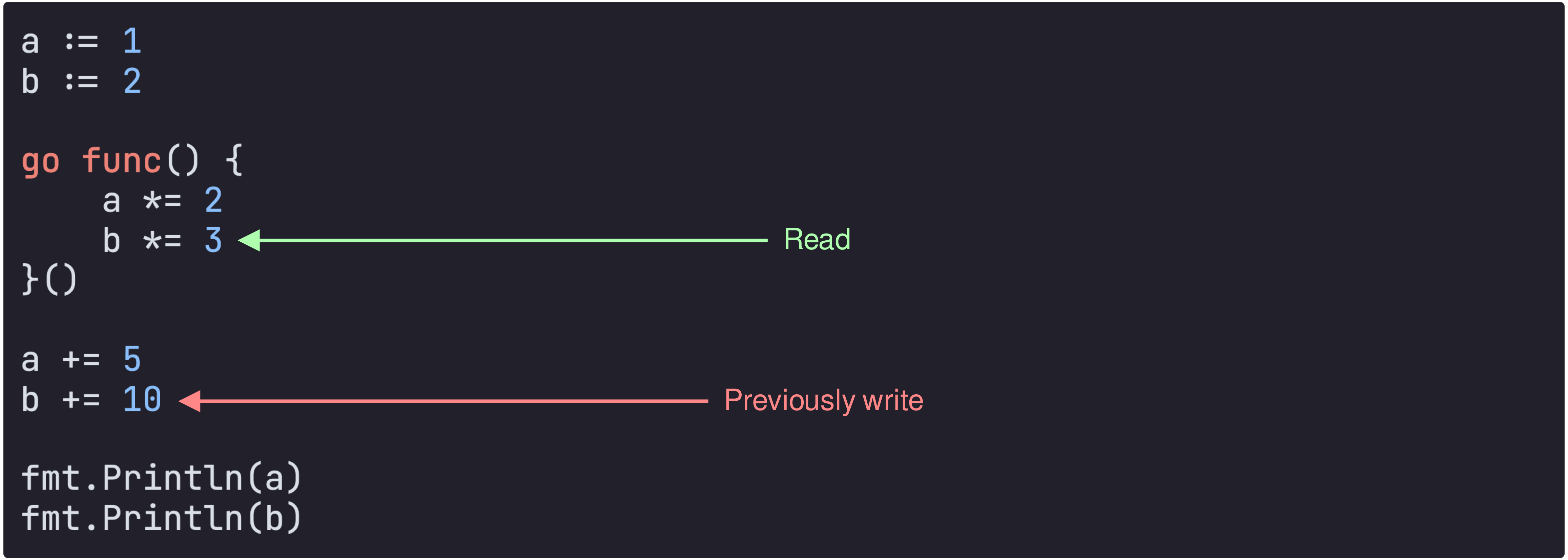


```go
a := 1
b := 2

go func() {
    a *= 2
    b *= 3
}()

a += 5
b += 10

fmt.Println(a)
fmt.Println(b)
```

Read

Previously write

Visual representation - 2/2



```go
a := 1
b := 2

go func() {
    a *= 2
    b *= 3
}()

a += 5
b += 10

fmt.Println(a)
fmt.Println(b)
```

Mutexes

- Use `Mutex` for exclusive access
- Use `RWMutex` for multiple readers, single writer access

Example

```
var mu sync.Mutex

mu.Lock()
// ^^^^
// Locks the mutex

mu.Unlock()
// ^^^^^^
// Releases the lock
```

```go
var mu sync.Mutex

a := 1
b := 2

mu.Lock()

go func() {
    mu.Lock()
    a *= 2
    b *= 3
    mu.Unlock()
}()

a += 5
b += 10

mu.Unlock()

fmt.Println(a)
fmt.Println(b)
```

Rules

- We can not access or modify shared data
- Data must be synchronized
- Otherwise data races can occur
- Always run with the race detector enabled in QA

When things go wrong

An example

Copying structs from the sync package - 1/2

```go
type User struct {
    lock sync.RWMutex
    Name string
}

func doSomething(u User) {
    u.lock.RLock()
    defer u.lock.RUnlock()
    // do something with `u`
}
```

```go
u := User{Name: "John"}
doSomething(u)
```

Copying structs from the sync package - 2/2

Issue

```
func doSomething(u User) {
//               ^^^^^^^
//               Copies User and the lock

    u.lock.RLock()
//  ^^^^^^^^^^^^^^
//  Locks a copy of the lock

    defer u.lock.RUnlock()
}
```

Solution

Use a pointer for `User` or `User.lock`

Other (common) issues

- Using `defer` in a loop
- Use `time.After` more than once
- Access global variables concurrently
- Accidental deadlock when using channels

Exercise 2

Summary

- Traverse through directories and sub-directories.
- Read and search for specified words within each file.
- Use a concurrent design to speed up the process.

Go to

training.brainhive.nl

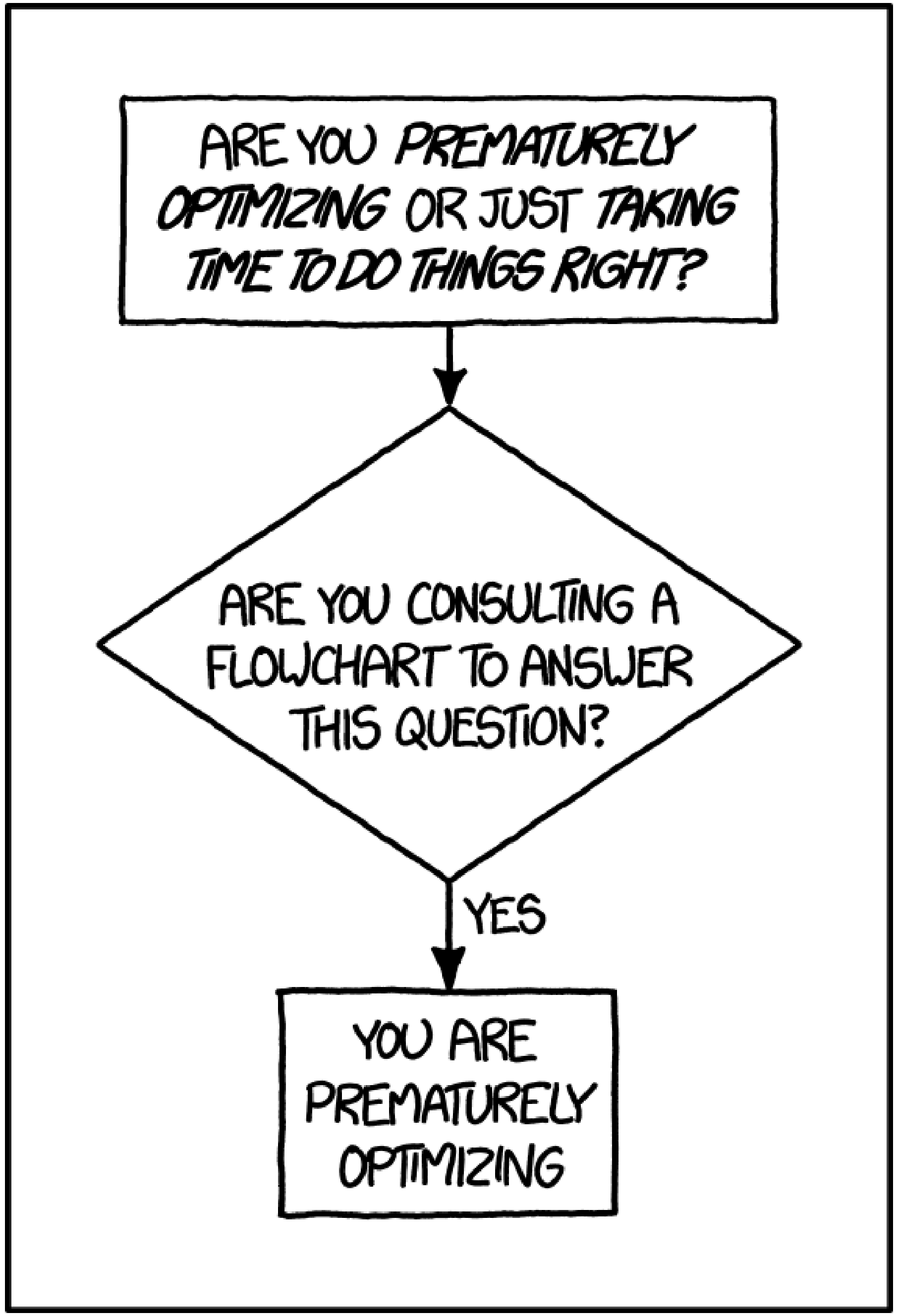
Recap

- What did we discuss?
- How are we doing?
- What's next?
    - Profiling
    - Error wrapping
    - Standard library
    - Exercise 3

Profiling

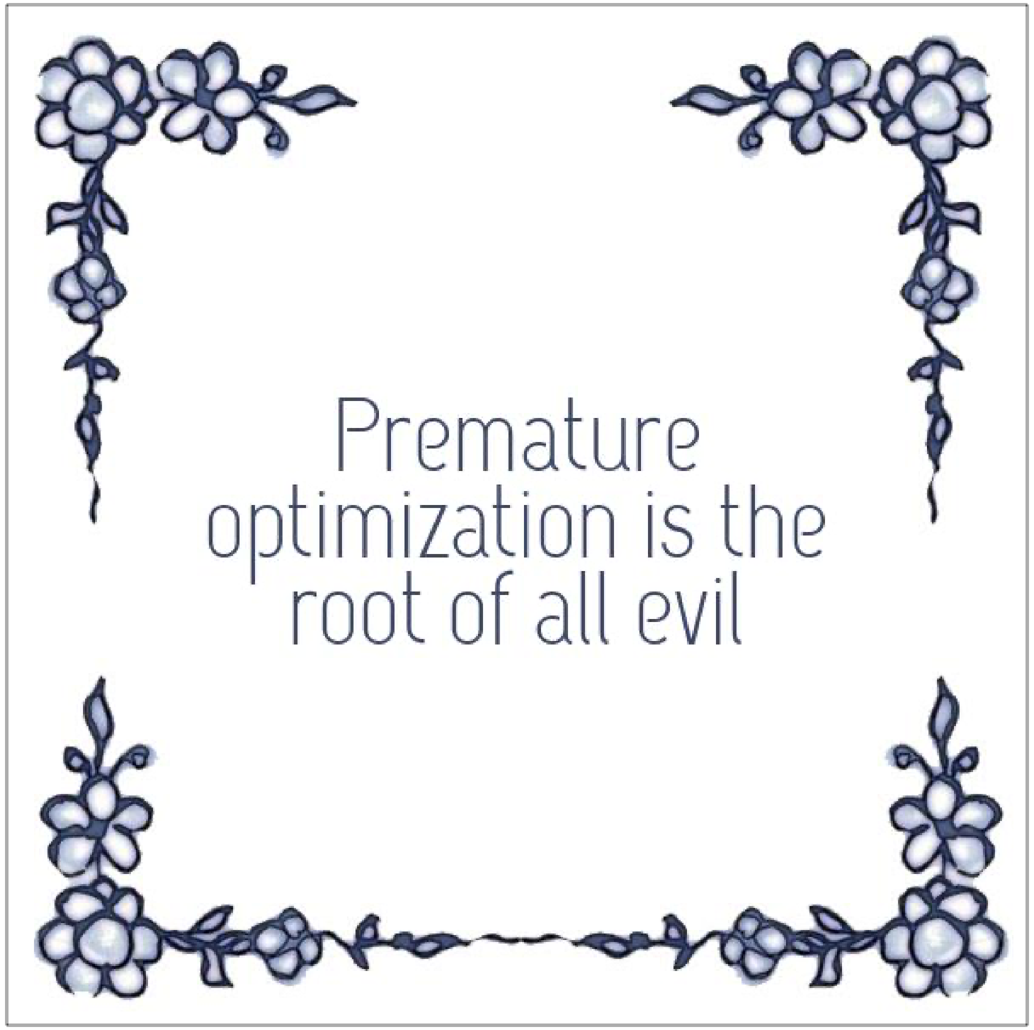
Why?

- Optimize performance
- Profiling = measuring performance of source code or executable
- Use profilers to understand program behavior


ARE YOU PREMATURELY OPTIMIZING OR JUST TAKING TIME TO DO THINGS RIGHT?
ARE YOU CONSULTING A FLOWCHART TO ANSWER THIS QUESTION?
YES
YOU ARE PREMATURELY OPTIMIZING

Benchmarking - 1/4

- You can't optimize what you can't measure
- Profile first, then optimize
- Example: optimize code that is not hot path



Benchmarking - 2/4

- Benchmarking is a form of profiling
- Go has builtin benchmarking framework
- Part of the testing framework
- Should be in `_test.go` file
- Prefixed with `Benchmark` (e.g. `BechmarkSignature`)

Benchmarking - 3/4

```
func BenchmarkBigLen(b *testing.B) {
    big := NewBig()

    b.ResetTimer()
    // ^^^^^^^^^
    // Resets timer, otherwise it includes setup time

    for i := 0; i < b.N; i++ {
    //              ^^^
    //              Number of iterations (is dynamic)
        big.Len()
    //  ^^^^^^^^^
    //  Function to benchmark
    }
}
```

Benchmarking - 4/4

- Always run test in a loop with `b.N` iterations
- Otherwise results are not reliable
- Number of iterations are based on the time
  - The faster it runs, the more iterations it requires
- Benchmark can be run with: `go test -bench .`
  - Where `.` means: "run all tests"

Go tools

- Go has builtin CPU/memory profiling
- CPU: `go test -bench . -cpuprofile cpu.out`
- Memory: `go test -bench . -memprofile mem.out`

Demonstration

- Testing
- Benchmarking
- Profiling

Exercise 3

Summary

- Use benchmarking/profiling to optimize the program
- This is a competition, the fastest implementation wins
- Good luck

Go to

training.brainhive.nl

Review

Standard library

Errors Package

- Contains helpers for creating and unwrapping errors
- Make sure to use `errors.Is` and `errors.As` to check for errors

Wrapping errors

```go
content, err := os.ReadFile("test.txt")
if err != nil {
    return fmt.Errorf("could not read file: %w", err)
}
```

```go
if errors.Is(err, os.ErrNotExist) {
    // Handle error
}
```

Parse errors - 1/3

- Last week we discussed using type assertions
- Using these assertions we can get the underlying error

Example

```go
_, err := os.ReadFile(":/hi<")
if pathErr, ok := err.(*fs.PathError); ok {
    fmt.Println(pathErr.Path) // Ok
}
```

Parse errors - 2/3

- Type assertions are not reliable as errors can be wrapped
- The underlying error will not be the custom error

Example

```go
err = fmt.Errorf("something went wrong: %w", err)

if pathErr, ok := err.(*fs.PathError); ok {
    fmt.Println(pathErr.Path) // Won't work as err is wrapped
}
```

Parse errors - 3/3

- How do we fix this?
- Use `errors.Is` and `errors.As` to check for errors

Solution

```
var pathErr *fs.PathError

if errors.As(err, &pathErr) {
    fmt.Println(pathErr.Path) // Ok
}
```

Error wrapping - 1/2

- Introduced in Go 1.13
    - `fmt.Errorf` supports `%w` to wrap errors
    - `errors.Is` / `errors.As` can be used to unwrap errors
    - `errors.Unwrap` can be used to unwrap errors

Interface

```go
type Wrapper interface {
    Unwrap() error
}
```

Error wrapping - 2/2

- `os.IsNotExist` / `os.IsExist` predates error wrapping
- They weren't changed as it would break compatibility promise
- These predicates will not unwrap errors properly
- Always use `errors.Is` / `errors.As` as it is more reliable

Example

```go
_, err := os.Stat("test.txt")

if errors.Is(err, fs.ErrNotExist) {
    fmt.Println(err) // Ok
}
```

Package / io

- Basic I/O primitives
- `Reader` and `Writer` interfaces
- Contains common errors such as: `EOF` or `ErrClosedPipe`

Interfaces

```go
type Reader interface {
    Read(buf []byte) (int, error)
}

type Writer interface {
    Write(buf []byte) (int, error)
}
```

Package / bufio

- Implements buffered I/O
- `Reader` and `Writer` interfaces
  - Wraps `io.Reader` / `io.Writer` to make it buffered
- Use `Scanner` to scan tokens in a reader

Example

```
scanner := bufio.NewScanner(strings.NewReader("hello world"))
scanner.Split(bufio.ScanWords)
scanner.Scan()

word := scanner.Text()
// Output: "hello"
```

Package / os

- Platform independent OS functionality
- `File` struct for file operations

Example

```go
file, err := os.Open("test.txt")
if err != nil {
        log.Fatal(err)
}

defer file.Close()
//    ^^^^^^^^^^
//    Make sure to close the file handle
```

Package / path/filepath

- Utilities for file paths which work with any OS
- Use `filepath.WalkDir` to recursively and deterministically search files

Example

```
fullpath := "/Users/john/Documents/test.txt"
dirname := filepath.Dir(fullpath)
// Output: /Users/john/Documents

newPath := filepath.Join(dirname, "new.txt")
// Output: /Users/john/Documents/new.txt

filename := filepath.Base(newPath)
// Output: new.txt
```

Package / strings - 1/2

- Functions which work with UTF-8 encoded strings
- Use `Builder` to efficiently build a string (minimizes memory copying)
- `Reader` implements `io.Reader` but for a string

Example

```
original := "hello foo"
value := strings.ReplaceAll(original, "foo", "bar")
value = strings.ToUpper(value)

fmt.Println(value)
// Output: HELLO BAR
```

Package / strings - 2/2

Don't use `strings.Compare` as its unoptimized by design

Comment from Go sourcecode

```
// NOTE(rsc): This function does NOT call the runtime cmpstring function,
                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// because we do not want to provide any performance justification for
// using strings.Compare. Basically no one should use strings.Compare.
                                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// As the comment above says, it is here only for symmetry with package bytes.
// If performance is important, the compiler should be changed to recognize
// the pattern so that all code doing three-way comparisons, not just code
// using strings.Compare, can benefit.
```

Package / bytes

- Similar to `strings` but for byte slices
- `Buffer` wraps a byte slice, and:
  - Implements both `io.Reader` and `io.Writer`
  - Contains some useful methods
- `Reader` implements `io.Reader` but for a byte slice
  - Also supports seeking

Example

```go
ok := bytes.Contains([]byte("hello world"), []byte("world"))
// Output: true
```

Package / fmt - 1/2

Formatted I/O with C-style printf/scanf functions

Verb	Summary
`%w`	Wraps error (can only be used as an `error` type)
`%v` / `%+v`	The value in a default format (`+` adds field names)
`%#v`	Go-syntax representation of the value
`%T`	Go-syntax representation of the type of the value
`%d`	Base 10 integers
`%f`	Floating point numbers

Package / fmt - 2/2

Example

```
branch := "main"
updateCount := 2
message := fmt.Sprintf("'%s' has %d pending updates", branch, updateCount)
// Output: 'main' has 2 pending updates
```

```
data := make(map[string]int)
data["foo"] = 100

fmt.Printf("data=%#v\n", data)
// Output: data=map[string]int{"foo":100}
```

Package / regexp

- Implements regular expression search
- Compile the regex once, and use it multiple times

Example

```
re := regexp.MustCompile(`var ([a-z]+)`)
//           ^^^^^^^^^^^
//           Panics if regex is invalid

matches := re.FindStringSubmatch("var foo int")

fmt.Println(matches[1])
// Output: foo
```

Package / time - 1/4

- Provides functionality for measuring and displaying time
- The `Time` type returned by `time.Now()` contains both a:
    - Wall clock: subject to changes for clock synchronization, for telling time
    - Monotonic clock: not subject to clock synchronization, for measuring time

Example

```
start := time.Now()
month := start.Month() // Type: time.Month (alias for int)
// Output: 1
```

Package / time - 2/4

Durations

```
start := time.Now()

time.Sleep(time.Second * 2)
//         ^^^^^^^^^^^^^^^
//         Accepts time.Duration as an argument

end := time.Now()
elapsed := end.Sub(start) // Type: time.Duration

sec := elapsed.Seconds()
// Output: 2
```

Package / time - 3/4

- Formatting time is done using `time.Format`
- Instead of using verbs like `%H`, Go uses a predefined layout
- Time layout uses the 2nd of january 2006 as a reference

Example

```
t := time.Now()
formatted := t.Format("2006-01-02 15:04:05")
// Output: 2023-02-01 10:05:00
```

Package / time - 4/4

Reference

```
Year: "2006" "06"
Month: "Jan" "January" "01" "1"
Day of the week: "Mon" "Monday"
Day of the month: "2" "_2" "02"
Day of the year: "__2" "002"
Hour: "15" "3" "03" (PM or AM)
Minute: "4" "04"
Second: "5" "05"
AM/PM mark: "PM"
```

Homework

Summary

- Convert from markdown to HTML
- Upgrade to an recursive and concurrent version

Go to

training.brainhive.nl