How to Go

What no one told me when I switched

Nice to Meet You

```
speaker := Speaker{
    Name: "Toly Rugalev",
    Age: 30,
    Experience: 12,
    ExperienceInGo: 6,
    NumberOfHands: 1,
    Position: "Senior Software Engineer",
    Company: "MessageBird",
}
```

My Journey with Go

- Switched to Go in 2017
- Still learning
- Learned the hard way

I wish I knew these things 6 years ago

Overview

1. Philosophy of Go
2. OOPn’t
3. ctx
4. panic(“DO NOT PANIC”)
5. P*inter&

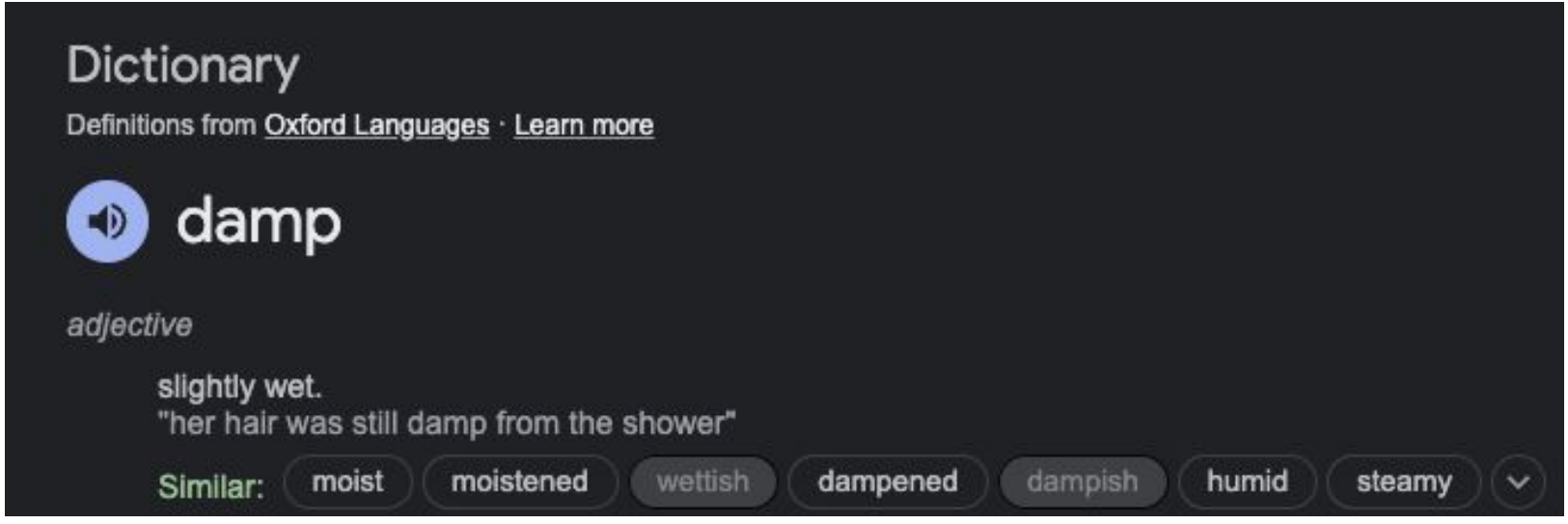
Philosophy: core principles

- Programming at scale
  - Fast compilation time
  - Dependency management
  - Cross-platform compatibility
- Minimalism
  - Easy to understand
    - Less mistakes
- Concurrent efficiency
  - Easy to build and understand
  - Easy to run
- Practicality
  - Solving real world problems
    - Testing
    - Networking
    - GC

Philosophy: not so DRY

- DRY (Don't Repeat Yourself)
- WET (Write Everything Twice)

DAMP (slightly WET)



OOPn’t

- Multi-paradigm
  - Procedural
  - Object Oriented
- Not OOP you used to
- Struct != Class
- Inheritance bad
- Composition good
- Implicit interface implementation

OOPn’t: Composition > Inheritance

```go
type Animal struct {
    name string
}

func (a *Animal) Speak() {
    fmt.Printf("%s makes a noise.\n", a.name)
}

type Dog struct {
    *Animal
}

func (d *Dog) Speak() {
    fmt.Printf("%s barks.\n", d.name)
}

func main() {
    a := &Animal{name: "Generic Animal"}
    d := &Dog{&Animal{name: "Fido"}}

    a.Speak() // "Generic Animal makes a noise."
    d.Speak() // "Fido barks."
```

OOPn’t: Implicit interface implementation

```go
type Shape interface {
    Area() float64
}

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}

type ColoredShape struct {
    Color string
    Shape
}

func main() {
    c := Circle{Radius: 5}
    cs := ColoredShape{Color: "red", Shape: c}
    fmt.Println("Colored Circle Area:", cs.Area())
}
```

Context

- Control flow statements
  - Request cancellations
  - Timeouts
- Cross-cutting values
  - Request info
  - Authentication
- Don’t overuse, but also don’t overthink
  - Authenticated user in context?
  - DB transaction in context?
- Always consider timeouts!
  - You will not be around when context.Background() ends (hint - heat death of the universe)

Context

```go
func main() {
    // Create a context with a timeout of 1 second
    ctx, cancel := context.WithTimeout(context.Background(), time.Second*1)
    defer cancel()

    // Start a long-running task
    go longRunningTask(ctx)

    // Wait for the context to be done
    <-ctx.Done()

    // Check if the context was cancelled or timed out
    if ctx.Err() == context.Canceled {
        fmt.Println("Context was cancelled")
    } else if ctx.Err() == context.DeadlineExceeded {
        fmt.Println("Context timed out")
    }
}
```

panic(“DO NOT PANIC”)

- Error is a value
- Forget exceptions
- Still need exceptions? Use errors + context + defer
- Only use `panic` as a last resort

```
func divide(x, y float64) (float64, error) {
    if y == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return x / y, nil
}
```

```
result, err := divide(4.0, 0.0)
if err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("Result:", result)
}
```

panic(“DO NOT PANIC”)

- init() functions
- The cause of a problem is in a source code, not data
- When there’s something developer forgot to do

Panics guide developers, errors guide users (or clients)

P*inter&

- Passing by value:
  - Value is copied
  - Thread-safe
  - (mostly) Garbage-free
- Passing by reference:
  - Pointer is copied
  - Thread-safety through locking (or lock-free data structures)
  - (most of the time) generates garbage
  - Requires escape analysis during compilation
- There’s no silver bullet
  - Or how my grand-grand mother said: depends on the ctx

P*inter&: decision making pointers

- If you care about performance
  - Avoid pointers, unless you have big sets of data
  - Most of the time copying one value is cheaper than a pointer
  - Write a benchmark and run it on a target platform
- If you care about safety
  - Avoid pointers at all costs
- If you don’t know/ don’t care
  - Then you should care about **safety**
- If you care about both
  - Sometimes pointers is a necessary evil
  - Concurrent access to the same data SHOULD be synchronized
  - Inspect and learn `atomic` and `sync` packages
  - Learn lock-free data structures (i.e. sync.Map)

P*inter&: decision making pointers

Performance = trading CPU cycles for memory and vice versa

Q&A