JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#13

STRUCTS & METHODS

BEYOND GROUPING VARIABLES: COMPOSITION, ALIGNMENT, AND STATE

The Anatomy of a Struct

More than a Container

A struct is a contiguous block of memory. The order of your fields actually matters for the CPU.

```
type Optimized struct {
    A int64 // 8 bytes
    B int32 // 4 bytes
    C bool // 1 byte
} // Minimal padding
```

Go aligns fields to their type's size. Grouping larger types at the top can reduce the total memory footprint of your struct by reducing "padding" bytes.

Designing for the Default

Make the Zero-Value Useful

In Go, variables are initialized to their "Zero-Value." A well-designed struct is ready to use immediately without a constructor.

```go
type Config struct {
    Mu    sync.Mutex
    Paths []string
}

var c Config
c.Mu.Lock() // Works immediately! No "NewConfig()" needed.
```

The Goal: If your struct requires a *New()* function to be safe, consider if you can redesign it so the zero-value "just works."

Composition over Inheritance

Struct Embedding

Go doesn't have *extends*. We use Embedding to promote fields and methods.

```go
type User struct {
    ID   int
    Name string
}

type Admin struct {
    User  // Embedded field
    Level int
}

a := Admin{}
fmt.Println(a.Name) // Promoted field access
```

Embedding is **not** inheritance. An *Admin* is not a *User*; it simply contains a *User*.
This is "Composition" at its finest.

Struct Tags (Metadata)

Communicating with the Outside World

Tags allow you to attach metadata to fields, used by encoders (JSON, DB, XML) via Reflection.

```go
type User struct {
    ID    int    `json:"id" db:"user_id"`
    Email string `json:"email,omitempty"`
}
```

Always use *omitempty* for optional fields to keep your API responses clean and efficient.

Methods - Functions with Context

Defining Behavior

A method is just a function with a Receiver argument. It creates a clear "Namespace" for behavior related to your data.

```go
type User struct {
    ID    int    `json:"id" db:"user_id"`
    Email string `json:"email,omitempty"`
}

func (u User) Greet() string {
    return "Hello, " + u.Name
}
```

The Logic: This allows your types to satisfy Interfaces (which we covered yesterday).

Opaque Types (Encapsulation)

Protecting the Internal State

```go
type Account struct {
    owner string // Lowercase = Unexported (Private)
    balance float64 // Lowercase = Unexported (Private)
}

// NewAccount acts as a "Gatekeeper"
func NewAccount(owner string) *Account {
    return &Account{owner: owner, balance: 0}
}

// Deposit is the ONLY way to change the balance
func (a *Account) Deposit(amount float64) {
    if amount > 0 { a.balance += amount }
}

// Balance is a "Getter" to read the private field
func (a *Account) Balance() float64 {
    return a.balance
}
```

By hiding the *balance* field, you guarantee that no external package can set a negative balance or bypass your business logic.

You can change the internal implementation of *Account* (e.g., changing *float64* to an *int64* for cents) without breaking any code that uses the Balance() method.

Rule of Thumb: Default to unexported fields. Only make them public if they are "Plain Old Data" (POD) with no logic attached.

The "New" Pattern

When you actually NEED a Constructor

Use a *New...* function when your struct cannot be used in its zero-state (e.g., it needs a network connection or a complex map initialization)

```go
func NewDatabase(url string) (*Database, error) {
    if url == "" {
        return nil, errors.New("url required")
    }
    return &Database{url: url}, nil
}
```

GOLANG /// BA

Recap:

- Align fields to save memory.
- Aim for useful zero-values.
- Use Composition (Embedding) over Inheritance.

Tonight we dive into the "Great Debate": Pointer vs. Value Receivers.