● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#11

FUNCTIONS: THE POWER OF MULTIPLE RETURNS

HOW GO’S MULTIPLE RETURN VALUES ELIMINATE THE "EXCEPTION" MESS

The Multiple Return Idiom

The Value-Error Pair

This is the heart of Go's error handling. Instead of trying/catching, we return the result and an error.

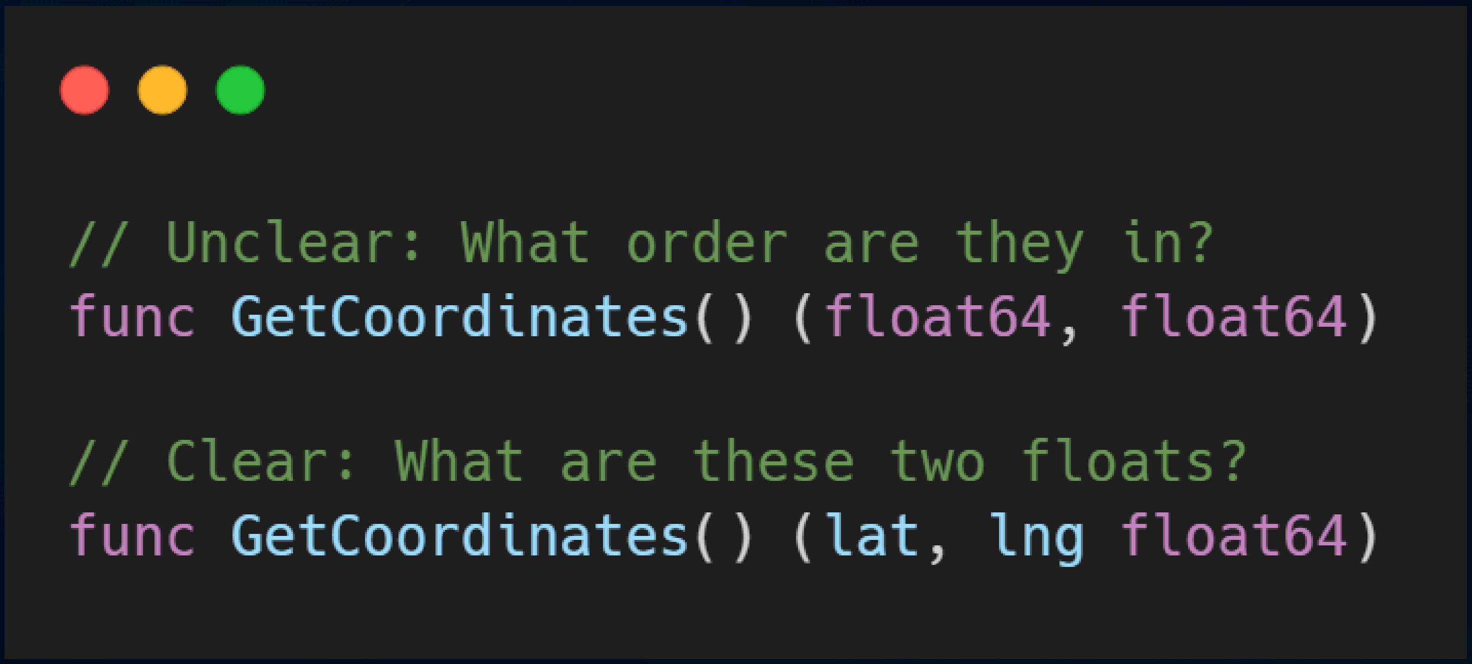
```go
func Divide(a, b float64) (float64, error) { // Returns the result and an error
    if b == 0 {
        return 0, errors.New("cannot divide by zero") // Returns a default value and the error
    }
    return a / b, nil // Returns the result and a nil error
}
```

The Golden Rule: Always return the error as the last value. This allows the caller to use the *if err != nil* pattern immediately, even forcing the caller to do so.

Named Return Parameters

A Self-Documenting Code

Use named return parameters specifically when you have multiple return values of the same type.



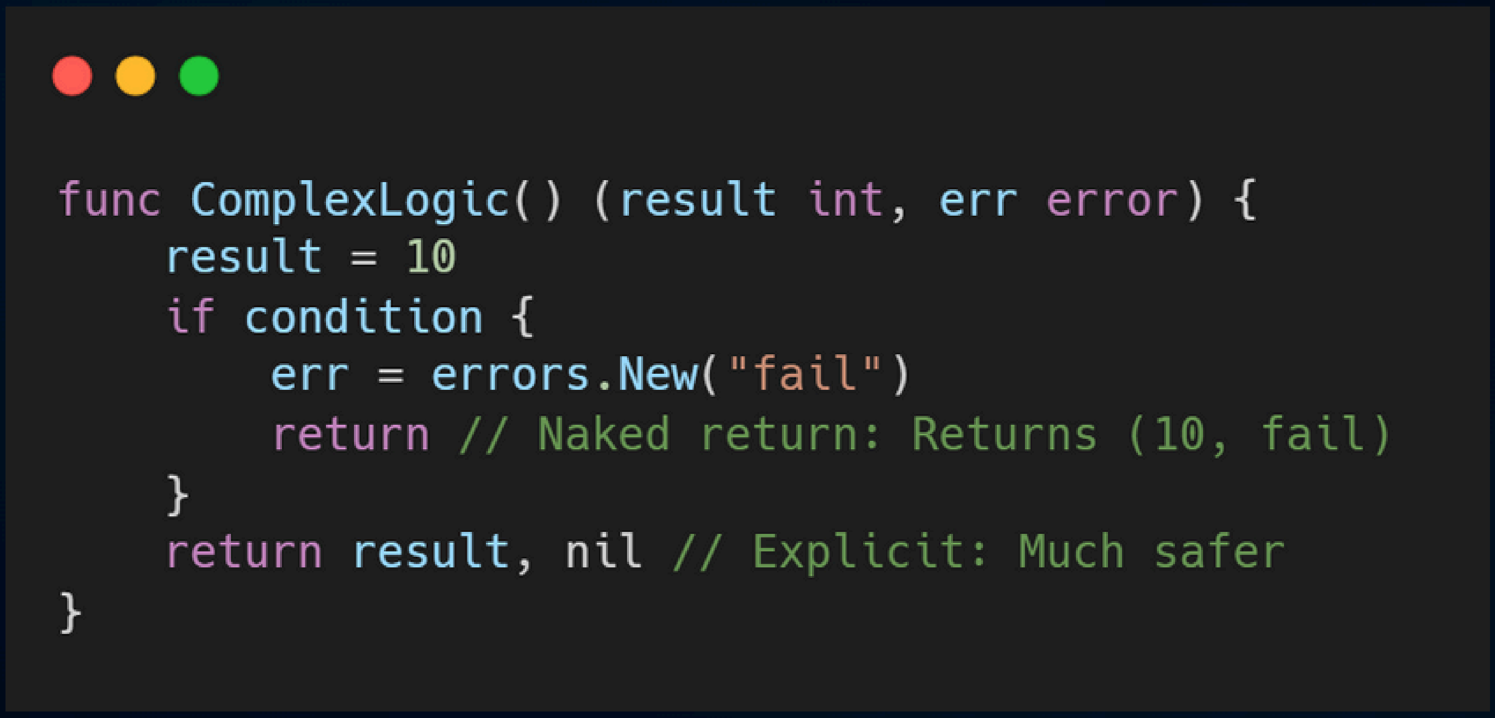
```go
// Unclear: What order are they in?
func GetCoordinates() (float64, float64)

// Clear: What are these two floats?
func GetCoordinates() (lat, lng float64)
```

The Benefit: It moves the documentation into the signature itself, making IDE autocompletion more helpful for other developers.

The Naked Return Trap

Why We Explicitly Return

When naming our return values Go initializes them to zero value and allows to return using a “naked” return to return them implicitly



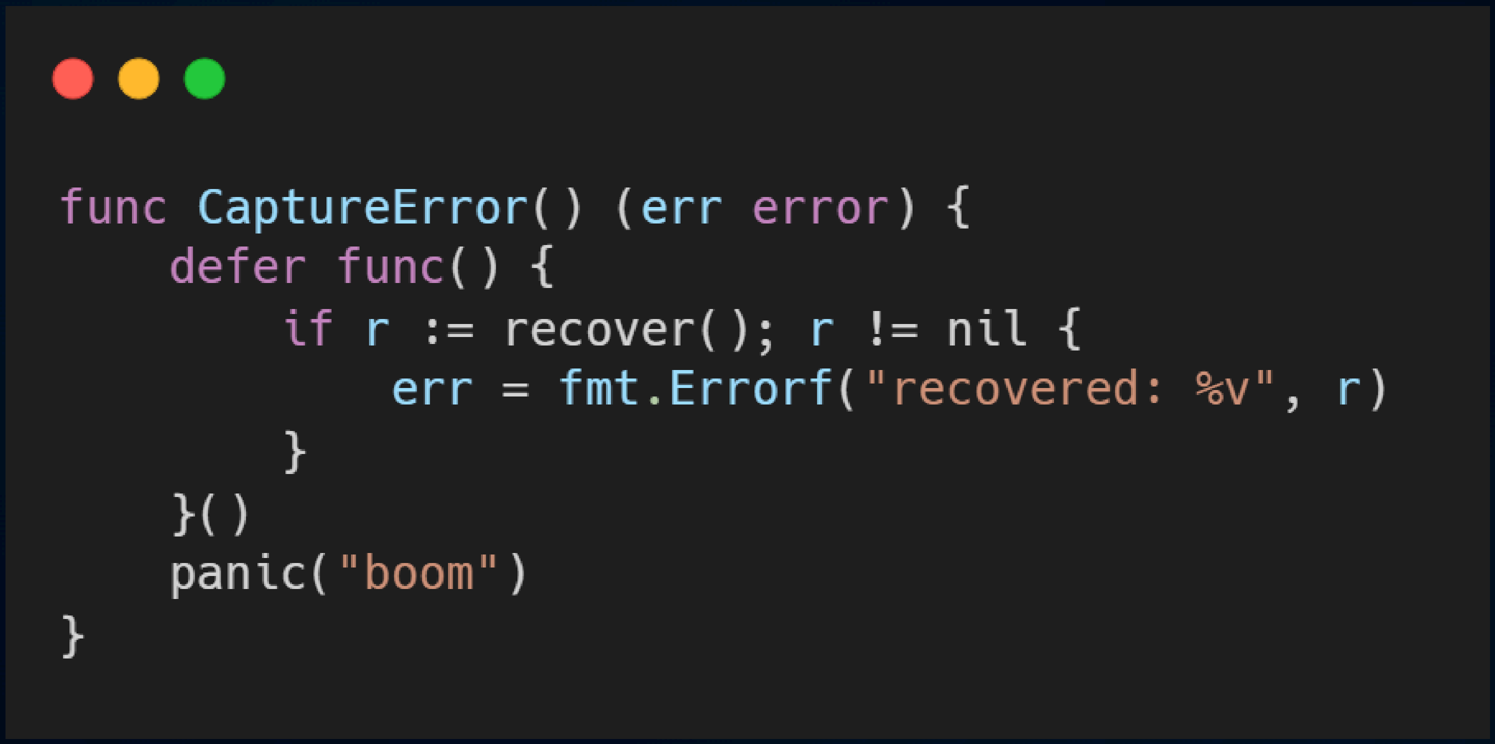
```go
func ComplexLogic() (result int, err error) {
    result = 10
    if condition {
        err = errors.New("fail")
        return // Naked return: Returns (10, fail)
    }
    return result, nil // Explicit: Much safer
}
```

Avoid Naked Returns. In long functions, a naked *return* makes it impossible to see what is being returned without scrolling back to the top.
The Rule: Always explicitly list your variables: *return result, err*. It's more verbose, but it prevents "shadowing" bugs and improves maintainability.

Named Returns & Defer

The One "Pro" Use Case: Defer

Named returns are the only way a *defer* block can modify a function's return value before it exits.



```go
func CaptureError() (err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("recovered: %v", r)
        }
    }()
    panic("boom")
}
```

*This is a specialized pattern used for recovery or wrapping errors in middleware.

Functions as First-Class Citizens

Higher-Order Functions

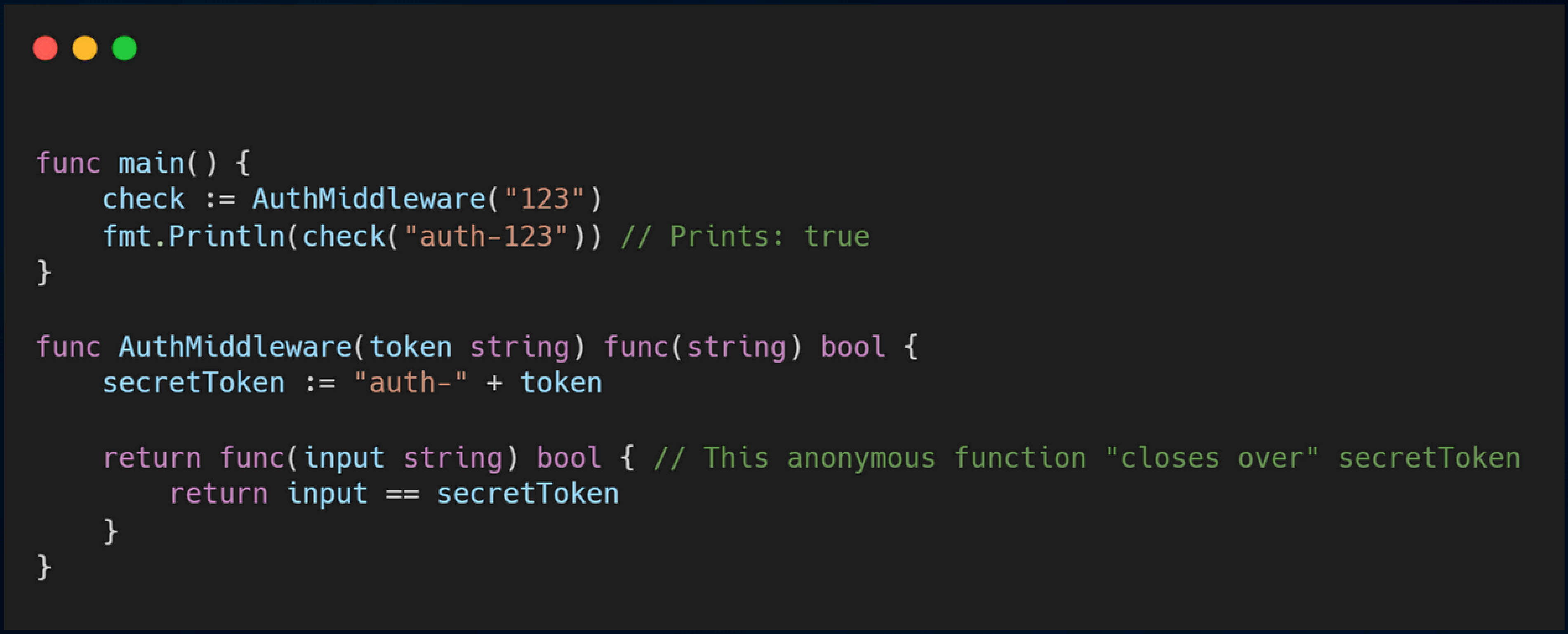
In Go, functions are values. You can pass them as arguments, which is the foundation of **Dependency Injection** in Go.

```go
type Filter func(int) bool

func ApplyFilter(nums []int, f Filter) []int {
    // ... logic
}
```

Anonymous Functions & Closures

Encapsulating Logic with Closures

A closure is an anonymous function that captures and carries variables from the scope where it was created. It's a powerful tool for Middleware and Stateful Factories.



```go
func main() {
    check := AuthMiddleware("123")
    fmt.Println(check("auth-123")) // Prints: true
}

func AuthMiddleware(token string) func(string) bool {
    secretToken := "auth-" + token

    return func(input string) bool { // This anonymous function "closes over" secretToken
        return input == secretToken
    }
}
```

State Without Globals: Closures allow you to maintain state (like configuration or DB handles) without using global variables.

Functional Options: This is the foundation of the "Functional Options" pattern used in gRPC and high-end Go libraries.

Memory Note: Variables captured by a closure live as long as the closure exists. Watch out for memory leaks in long-running goroutines!

Function Signatures as Contracts

API Consistency

Your function signature is a contract.
Keep the order consistent. If your package uses (data, error), don't switch to (error, data) for one function. It violates the "**Principle of Least Astonishment**".

Clean Code, Clear Returns. Recap:

- Use Named Returns for docs, but avoid Naked Returns.
- Multiple returns eliminate the need for Exceptions.
- Use closures to encapsulate state cleanly.

Tonight we dive into Interfaces: Why you should accept interfaces but return structs.