● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#14

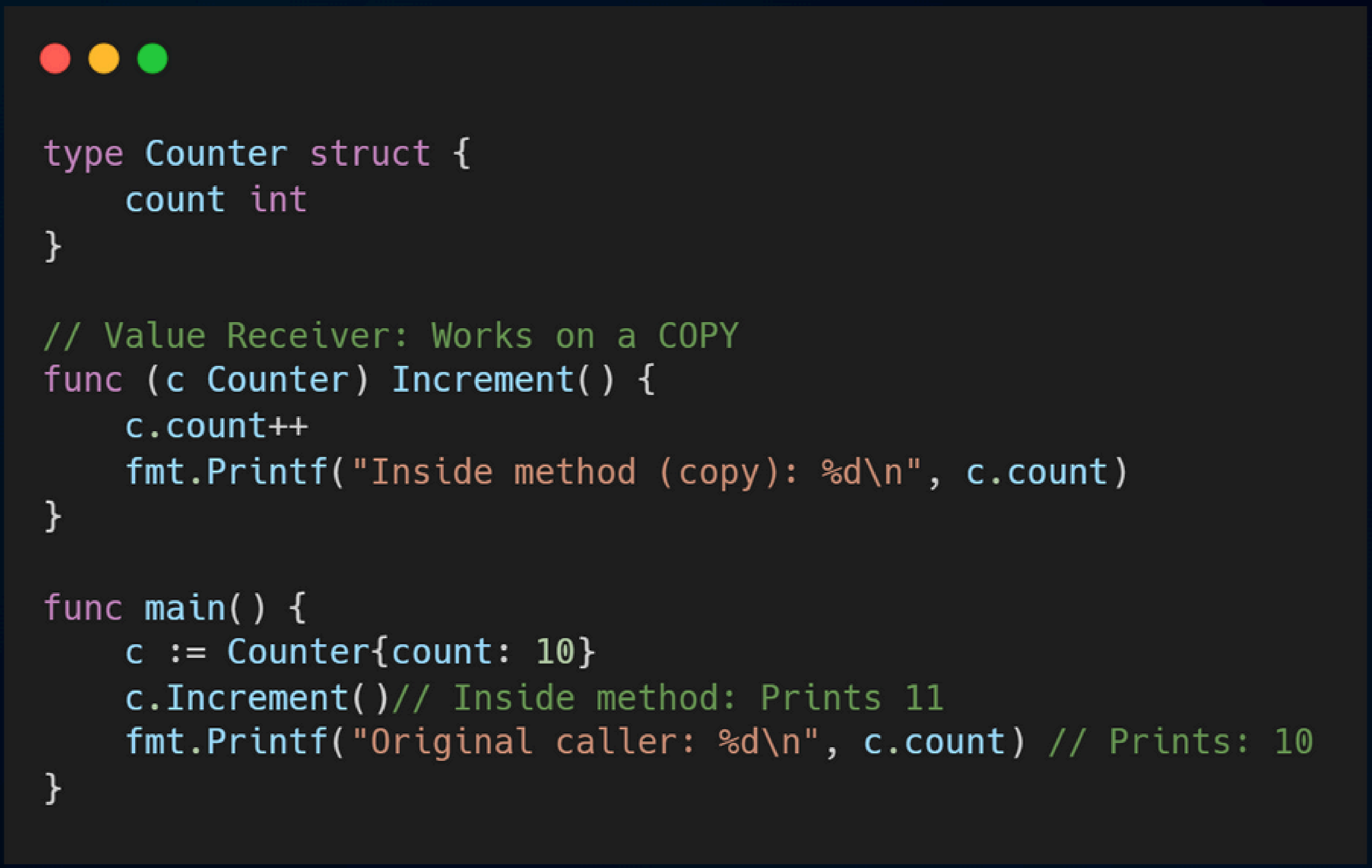
POINTER VS. VALUE RECEIVERS

CHOOSING THE RIGHT RECEIVER FOR YOUR METHODS

Value Receivers (The Copy)

The Safety of the Value Receiver

When you use a value receiver, Go creates a full copy of the struct before running the method.



```go
type Counter struct {
    count int
}

// Value Receiver: Works on a COPY
func (c Counter) Increment() {
    c.count++
    fmt.Printf("Inside method (copy): %d\n", c.count)
}

func main() {
    c := Counter{count: 10}
    c.Increment()// Inside method: Prints 11
    fmt.Printf("Original caller: %d\n", c.count) // Prints: 10
}
```

Intentional Immutability: Use value receivers when you want to keep state.
The Copy Cost: If your struct contains a lot of data, this copy happens on every single call which can lead to unnecessary CPU cycles and memory pressure.
The Thread-Safety Bonus: Since every call gets its own isolated copy, you don't need a *sync.Mutex* to protect the data from concurrent reads within the method itself.

Pointer Receivers (The Share)

Modifying State in Place

A pointer receiver passes the memory address. This allows the method to mutate the original struct.

```go
func (u *User) UpdateName(newName string) {
    u.Name = newName // Changes the original caller's data
}
```

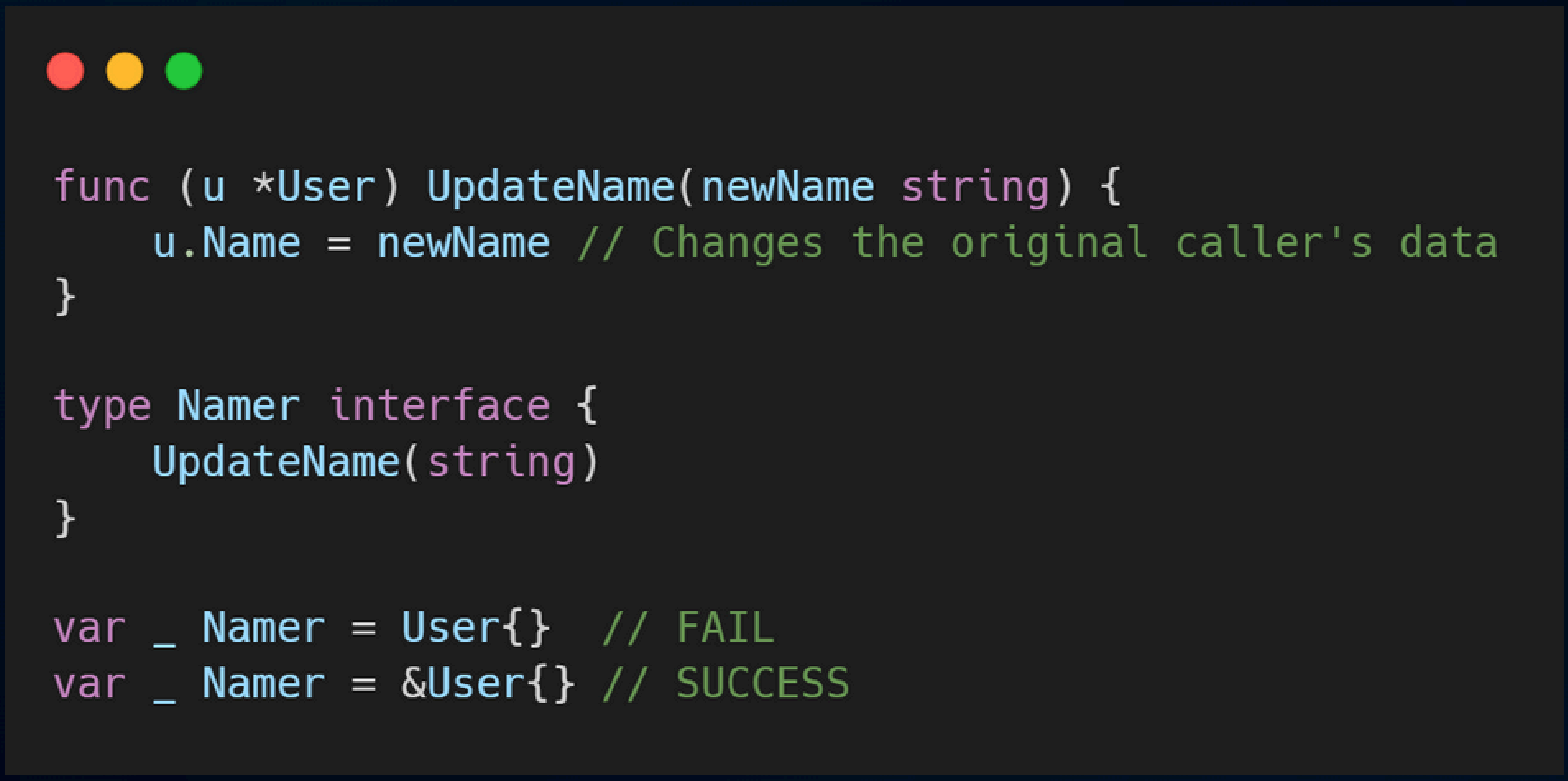
Even if the struct is 10MB, only 8 bytes (the pointer) are copied.

The Interface Trap

Method Sets and Interfaces

This is where the choice becomes critical for your architecture.

The Rule: If a method has a pointer receiver, only a pointer to the struct satisfies an interface.



```go
func (u *User) UpdateName(newName string) {
    u.Name = newName // Changes the original caller's data
}

type Namer interface {
    UpdateName(string)
}

var _ Namer = User{}  // FAIL
var _ Namer = &User{} // SUCCESS
```

Copying vs. Escaping

The Myth of the "Faster Pointer"

Many developers use pointers thinking they are "saving memory".

Copying a small struct (like 2-3 fields) is often faster than passing a pointer.

- Pointers can cause the struct to Escape to the Heap, increasing work for the Garbage Collector.
- Small values stay on the Stack, which is essentially "free" to clean up.

The Rule of Consistency

Don't Mix and Match

If your struct needs even one pointer receiver (because it mutates state), you should generally make all methods on that struct use pointer receivers.

Why? It prevents confusion for the caller and ensures that the entire "Method Set" of the struct is consistent when satisfying interfaces.

Nils and Panics

Handling the Nil Receiver

In Go, you can actually call a method on a nil pointer.

```go
func (u *User) GetName() string {
    if u == nil { return "Guest" }
    return u.Name
}

var u *User
fmt.Println(u.GetName()) // Prints "Guest" instead of panicking!
```

This is a great way to provide default behaviors for optional data without crashing the program.

The Decision Matrix

When to choose which?

Use a Value Receiver if:

- The struct is small (e.g., coordinates, timestamps).
- The struct is a basic "data holder" (POD).
- You want to guarantee immutability.

Use a Pointer Receiver if:

- The method needs to modify the receiver.
- The struct contains a *sync.Mutex* (you must never copy a Mutex!).
- The struct is large and copying it would be expensive.

Master the Receiver. Recap:

- Value receivers are for safety and small data.
- Pointer receivers are for state changes and large data.
- Don't mix them on the same struct!

Tonight we dive deeper into Pointers , memory and Garbage Collector pressure