JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#17

INTERFACES - BEHAVIOR OVER INHERITANCE

WHY GO'S "IMPLICIT" INTERFACES CHANGE EVERYTHING

The Implicit Contract

If it walks like a duck...

In other languages, you must explicitly declare that a class implements an interface. In Go, it’s automatic. If it implements – it implements.

```go
type Logger interface {
    Log(message string)
}

type ConsoleWriter struct{}

// ConsoleWriter satisfies Logger automatically just by having the method.
func (cw ConsoleWriter) Log(m string) {
    fmt.Println(m)
}
```

This allows you to define interfaces for code you don't even own (like third-party libraries), enabling true decoupling.

Interface as a Behavior Filter

Small is Beautiful

The most powerful interfaces in Go are the smallest ones: *io.Reader*, *io.Writer*, *error*.

```go
// 1. io.Reader: The universal way to get data IN
type Reader interface {
    Read(p []byte) (n int, err error)
}

// 2. io.Writer: The universal way to push data OUT
type Writer interface {
    Write(p []byte) (n int, err error)
}

// 3. error: The universal way to communicate FAILURE
type error interface {
    Error() string
}
```

Universal Compatibility: Because **os.File*, **bytes.Buffer*, and **net.Conn* all implement *Read* and *Write*, you can swap a disk-file for a network-socket without changing a single line of your business logic.

Refactoring Hint: If your interface has more than 3 methods, ask yourself: "Can I break this into smaller pieces and compose them later?"

"The bigger the interface, the weaker the abstraction" - Rob Pike.

The Golden Rule of Go Design

Accept Interfaces, Return Structs

Accept Interfaces: Make your functions flexible.

Return Structs: Give the caller the full power of the concrete type.

```go
// GOOD: I can pass a File, a Buffer, or a Network Conn
func SaveData(w io.Writer, data []byte)

// BAD: I can ONLY pass a pointer to an os.File
func SaveData(f *os.File, data []byte)
```

Discovering Interfaces

Don't Design Interfaces Upfront

In Java, you often write the Interface before the Class. In Go, we do the opposite.

The Senior Process:

1. Write the concrete implementation (Struct).
2. Write a second implementation.
3. Discover the common behavior and extract the interface.

The Benefit: This prevents "Interface Pollution" – creating abstractions that you never actually end up using.

The "any" (Empty Interface) Trap

With great power comes ZERO type safety!

interface{} (now *any*) can hold any value, but it tells you nothing about behavior.

Warning: If your function accepts *any*, you are bypassing the compiler. Use it only for truly generic data (like JSON parsing or logging).
If you use it for business logic, you're likely losing the "Go way".

Interface Satisfaction Check

The "Static" Compliance Trick

How do you guarantee a struct implements an interface at compile time?

```go
type MyInterface interface {
    DoWork()
}
type MyStruct struct{}

// This line will fail to compile if MyStruct
// doesn't implement MyInterface.
var _ MyInterface = (*MyStruct)(nil)
```

Use this in your *internal* packages to ensure that a refactor doesn't accidentally break your interface implementation before you run your tests.

Composition via Interfaces

Building Complex Systems

You can embed interfaces into structs or even other interfaces.

```
type ReadWriter interface {
    io.Reader
    io.Writer
}
```

This allows you to build powerful abstractions by combining small, focused behaviors into larger ones.

+

Behavior > Identity

Recap:

- Interfaces are satisfied implicitly.
- Accept Interfaces (Inputs), Return Structs (Outputs).
- Keep interfaces small and focused.

Have you ever created an interface only to realize you only had one struct implementing it? Was it worth it for the testing? Let’s talk! 👇