Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 8

Interfaces

You have been writing Go code for several chapters now, and you have already brushed against interfaces — `fmt.Stringer`, `io.Reader`, `error`. This chapter is where they become first-class citizens. Go interfaces are the primary tool for abstraction, and they work very differently from Java's. Java uses **nominal** interfaces: a class is a `Comparable` only if its source says `implements Comparable`, so retrofitting a type you do not own means editing (or wrapping) its source. Go uses **structural** typing: a type satisfies an interface the moment it has the right methods, no declaration required, which lets you define a tiny interface in your own package and have third-party types satisfy it for free. That decoupling is what makes Go code so easy to mock in tests and extend without inheritance. Understanding interfaces unlocks idiomatic Go: clean, testable, composable code that does not require a class hierarchy.

Implicit Interface Satisfaction

In Java, a class declares that it implements an interface:

```
class Song implements Stringer { ... }
```

If you forget `implements`, the class is not a `Stringer`, even if it has every required method. The compiler checks the declaration, not the methods.

Go flips this completely. There is no `implements` keyword. Any type that has the required methods **automatically** satisfies the interface. No declaration needed.

Here is the `fmt.Stringer` interface from the standard library:

```
type Stringer interface {
    String() string  // returns a human-readable representation of the value
}
```

Any type with a `String() string` method is automatically a `fmt.Stringer`.

```
type Track struct {
    Title  string
    Artist string
    BPM    int
}

func (t Track) String() string {
    return fmt.Sprintf("%s by %s (%d BPM)", t.Title, t.Artist, t.BPM)
}
```

`Track` is now a `fmt.Stringer`. No annotation, no registration. `fmt.Println` calls `String()` automatically when it sees a `Stringer`:

```
t := Track{Title: "Sounds of Slashdot", Artist: "San Mehat", BPM: 144}
fmt.Println(t) // Sounds of Slashdot by San Mehat (144 BPM)
```

Tip: This is called **structural typing** or **duck typing** with compile-time checking. If it has the right methods, it satisfies the interface — the compiler verifies this at the point of use, not at the point of definition. Java's approach is **nominal typing**: the name of the interface in the declaration is what matters, not the shape of the type.

Checking Satisfaction Explicitly

You will sometimes want to assert at compile time that a type satisfies an interface, without actually using the interface in a function call. The idiom is:

```
var _ fmt.Stringer = Track{}  // compile error if Track does not satisfy fmt.Stringer
```

The blank identifier discards the value; the assignment only exists to force a compile-time check. This is useful at the top of a file as documentation and a safety net.

Pointer Receivers and the Method Set

There is a wrinkle that trips up Java programmers, who are used to every method being dispatched on a reference. If a method has a **pointer receiver** (Chapter 6), only `*T` satisfies the interface, not `T`. The value type `T` is missing that method from its method set, so it does not satisfy the interface.

```
type Counter struct{ n int }

func (c *Counter) String() string { return fmt.Sprintf("count=%d", c.n) }

var _ fmt.Stringer = &Counter{}  // OK: *Counter has String()
var _ fmt.Stringer = Counter{}   // compile error: Counter does not implement fmt.Stringer
                                 // (method String has pointer receiver)
```

The fix is to pass the address (`&Counter{}`) or to give the method a value receiver if it does not need to mutate. See Chapter 6 for the full rules on value versus pointer receivers and method sets.

Interface Composition

Go interfaces can embed other interfaces, combining their method sets. The standard library uses this pervasively.

```
// io.Reader requires one method
type Reader interface {
    Read(p []byte) (n int, err error)  // reads up to len(p) bytes into p
}

// io.Writer requires one method
type Writer interface {
    Write(p []byte) (n int, err error)  // writes len(p) bytes from p
}

// io.ReadWriter is both
type ReadWriter interface {
    Reader  // embeds io.Reader
```

```
    Writer  // embeds io.Writer
}
```

A type that implements both `Read` and `Write` automatically satisfies `io.ReadWriter`.

You can define your own composed interfaces the same way:

```
type ReadWriteCloser interface {
    io.Reader   // Read(p []byte) (n int, err error)
    io.Writer   // Write(p []byte) (n int, err error)
    io.Closer   // Close() error
}
```

In Java, you would write `interface ReadWriteCloser extends Reader, Writer, Closeable`. Go's embedding syntax is a bit cleaner — you list the interfaces you want to include, and the compiler assembles the combined method set.

Tip: Prefer small, single-method interfaces over large ones. The `io` package is the gold standard: `Reader`, `Writer`, `Closer`, `Seeker` are each one method. Larger interfaces emerge from composing small ones. Do not define an interface until you have a concrete use case that requires it — premature abstraction adds indirection without benefit. [*no-premature-interface*]

any — The Top Type

Sometimes you want a variable that can hold any type. In Java, you do this with `Object` — `Object v` means any type of object can be assigned to `v`. You can do the same in Go with `var v interface{}`, but that is a rather verbose way to express that `v` can be assigned anything. So, Go 1.18 introduced `any` as an alias for `interface{}`. They are identical; `any` is just friendlier to read.

```
var x any = 42      // x holds an int
x = "sabor a mí"    // now x holds a string
x = []int{1, 2, 3}  // now x holds a slice
```

`any` is Go's counterpart to Java's `Object` — every type satisfies the empty interface because there are no methods to implement.

Wut: `any` is not a magic box that avoids copies. Assigning a value to an `any` variable wraps it in an interface value, which holds the concrete type plus a pointer to the data. Despite folklore to the contrary, modern Go does **not** store small values like `int` or `bool` inline in the interface; that inline optimization was removed back in Go 1.4, so a non-pointer value is boxed. The runtime may avoid the allocation in some cases (for example, small integers it keeps cached), but it never stores the value itself inline in the interface word.

Older Go code uses `interface{}` everywhere. When you read code that predates Go 1.18, `interface{}` and `any` mean exactly the same thing. New code should use `any`.

Tip: Use `any` sparingly. Code that traffics in `any` values gives up compile-time type safety and often requires type assertions (see below) to get the value back out. Generics (Chapter 18) are usually the better choice when you want a function that works with multiple types.

Type Assertions

An **interface value** holds two things: the concrete type and the concrete value. A type assertion extracts the concrete value.

The Panicking Form

```
var i any = "Saltwater"

s := i.(string)      // assert that i holds a string; assign it to s
fmt.Println(s)       // Saltwater
```

If the assertion is wrong, the program panics immediately:

```
n := i.(int)  // panic: interface conversion: interface {} is string, not int
```

Use this form when you are certain of the type — for example, immediately after a type switch case.

The Safe Form

```
s, ok := i.(string)  // ok is true if i holds a string
if ok {
    fmt.Println("got string:", s)
} else {
    fmt.Println("not a string")
}
```

The safe form never panics. If the type does not match, `ok` is `false` and `s` is the zero value of the asserted type.

Trap: Always use the two-value form (`v, ok := i.(T)`) when you are not certain of the type. The single-value form panics on a wrong guess, which is a runtime crash, not a compile error.

Type Switches

A **type switch** is a `switch` statement that dispatches on the dynamic type of an interface value. Chapter 4 showed a preview; here is the full picture.

```
func describe(i any) string {
    switch v := i.(type) {
    case int:
        return fmt.Sprintf("int: %d", v)      // v is int here
    case string:
        return fmt.Sprintf("string: %q", v)   // v is string here
    case bool:
        return fmt.Sprintf("bool: %t", v)     // v is bool here
    case []int:
        return fmt.Sprintf("[]int of length %d", len(v))
    case nil:
        return "nil"
    default:
        return fmt.Sprintf("unknown type: %T", v)
    }
}
```

The expression `i.(type)` is only valid inside a type switch — you cannot write it anywhere else. In each case, `v` is automatically converted to the matched concrete type. In the `default` case, `v` retains the type of `i` (i.e., `any`).

You can match multiple types in one case:

```go
switch v := i.(type) {
case int, int64:
    fmt.Println("some integer:", v)  // v is any here because the types differ
case string:
    fmt.Println("string:", v)
}
```

When a case lists more than one type, `v` takes the type of the switch expression (here `any`) because the compiler cannot assign a single concrete type to `v`.

Tip: Type switches are the idiomatic Go replacement for Java's `instanceof` chains. In Java you write:
`if (obj instanceof String s) { ... }`
`else if (obj instanceof Integer n) { ... }`
In Go you write a type switch. It is cleaner and exhaustive — the `default` case catches everything else.

Key Standard Library Interfaces

Go's standard library defines a small set of interfaces that appear everywhere. Knowing them lets you understand most Go code at a glance.

`io.Reader`

```go
type Reader interface {
    Read(p []byte) (n int, err error)  // reads into p; returns bytes read and error
}
```

`Read` fills the slice `p` with up to `len(p)` bytes. It returns the number of bytes actually read and any error. When the underlying data is exhausted, it returns `0, io.EOF`. A reader may also return the final `n > 0` bytes together with `io.EOF` in the same call, so always process the `n` bytes you got before acting on the error.

Here is a concrete type that implements `io.Reader`:

```go
// CountReader wraps an io.Reader and counts bytes as they are read.
type CountReader struct {
    r     io.Reader
    count int
}

func (cr *CountReader) Read(p []byte) (int, error) {
    n, err := cr.r.Read(p)  // delegate to the underlying reader
    cr.count += n           // tally the bytes
    return n, err
}
```

Any function that accepts an `io.Reader` will accept `*CountReader` without modification.

`io.Writer`

```go
type Writer interface {
    Write(p []byte) (n int, err error)  // writes all of p; returns count and error
}
```

`Write` must write exactly `len(p)` bytes or return an error. `os.Stdout`, `*os.File`, `*bytes.Buffer`, and `*strings.Builder` all satisfy `io.Writer`.

A minimal implementation:

```
// UpperWriter wraps an io.Writer and converts all bytes to upper case.
type UpperWriter struct {
    w io.Writer
}

func (uw *UpperWriter) Write(p []byte) (int, error) {
    upper := bytes.ToUpper(p)  // convert to upper case
    return uw.w.Write(upper)   // delegate to the underlying writer
}
```

`fmt.Stringer`

```
type Stringer interface {
    String() string  // returns a human-readable string representation
}
```

`fmt.Println`, `fmt.Sprintf` with `%v` or `%s`, and most other `fmt` functions check whether a value satisfies `Stringer` and call `String()` if so. You saw this at the start of the chapter with `Track`.

`error`

```
type error interface {
    Error() string  // returns the error message
}
```

`error` is a predeclared interface, not a type in any package — it is part of the language itself. Any type with an `Error() string` method is an `error`. Error handling is covered in full in Chapter 9; for now, just note the shape.

```
type ValidationError struct {
    Field   string
    Message string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("validation failed on %s: %s", e.Field, e.Message)
}
```

`*ValidationError` is now an `error` and can be returned anywhere an `error` is expected.

`sort.Interface`

```
type Interface interface {
    Len() int              // returns the number of elements in the collection
    Less(i, j int) bool    // reports whether element i should sort before element j
    Swap(i, j int)         // swaps the elements at positions i and j
}
```

Any type that implements all three methods can be sorted by `sort.Sort`. The standard library does not know or care what you are sorting — it only calls these three methods.

Here is a concrete type that sorts a slice of songs by title:

```
type Song struct {
    Title  string
    Artist string
```

```
}

// ByTitle wraps a []Song and sorts it alphabetically by title.
type ByTitle []Song

func (b ByTitle) Len() int           { return len(b) }
func (b ByTitle) Less(i, j int) bool { return b[i].Title < b[j].Title }
func (b ByTitle) Swap(i, j int)      { b[i], b[j] = b[j], b[i] }
```

Use it with `sort.Sort`:

```
songs := []Song{
    {Title: "The Sound of Silence", Artist: "Disturbed"},
    {Title: "Sandstorm",            Artist: "Darude"},
    {Title: "Better Off Alone",     Artist: "DJ Cobra"},
}

sort.Sort(ByTitle(songs))

for _, s := range songs {
    fmt.Printf("%s --- %s\n", s.Title, s.Artist)
}
// Better Off Alone --- DJ Cobra
// Sandstorm --- Darude
// The Sound of Silence --- Disturbed
```

`ByTitle(songs)` is a type conversion that reinterprets the `[]Song` slice as a `ByTitle` — no copying occurs.

Tip: `sort.Interface` is best suited to types that need to be sortable as a **first-class capability** — for example, a domain type that your package exposes and that users will sort repeatedly or pass to generic sort utilities. For a one-off sort inside a single function, Go 1.21's `slices.SortFunc` is simpler:

```
slices.SortFunc(songs, func(a, b Song) int {
    return cmp.Compare(a.Title, b.Title)  // cmp package, Chapter 7
})
```

You saw `slices.SortFunc` and `cmp.Compare` in Chapter 7 (remember to `import "cmp"`); use `sort.Interface` when the sortable behavior belongs to the type itself.

Accept Interfaces, Return Structs

One of Go's most important design idioms is: **accept interfaces, return concrete types**.

When a function's parameter is an interface, the caller can pass any type that satisfies it — including a mock in a test. When a function returns a concrete type, callers get access to all the methods of that type, not just the interface subset. Define the interface in the package that consumes it, not in the package that provides the implementation. [*interface-in-consumer*]

```
// Good: accepts io.Writer so any writer will do --- os.Stdout, a file, a buffer
func WriteJSON(w io.Writer, v any) error {
    enc := json.NewEncoder(w)
    return enc.Encode(v)
}

// Less flexible: only accepts *os.File
func WriteJSONToFile(f *os.File, v any) error {
```

```
    enc := json.NewEncoder(f)
    return enc.Encode(v)
}
```

Returning interfaces makes things harder:

```
// Avoid: callers cannot use *bytes.Buffer methods --- only io.Writer methods
func NewBuffer() io.Writer {
    return &bytes.Buffer{}
}

// Better: callers get the full *bytes.Buffer API
func NewBuffer() *bytes.Buffer {
    return &bytes.Buffer{}
}
```

Tip: Accept the narrowest interface that meets your needs. Return the most specific concrete type you can. This maximizes caller flexibility and testability without hiding useful API. [*return-concrete-types*]

The rule has one well-known exception: the `error` interface. Functions return `error` (an interface) rather than a concrete error type so that callers are not coupled to the specific error implementation. Chapter 9 explains why.

The Interface Nil Trap

This is one of Go's most notorious gotchas. Read this section carefully.

An interface value has two components: a **type** and a **value**. An interface is `nil` only when **both** the type and the value are `nil`. A typed nil — a nil pointer stored in an interface — is **not** nil.

```
type MyError struct{ msg string }

func (e *MyError) Error() string { return e.msg }

func mayFail(fail bool) error {
    var e *MyError  // e is a typed nil: type=*MyError, value=nil
    if fail {
        e = &MyError{msg: "something went wrong"}
    }
    return e        // BUG: always returns a non-nil interface!
}

func main() {
    err := mayFail(false)
    if err != nil {
        fmt.Println("got error:", err)  // this line executes even when fail=false!
    }
}
```

When `fail` is `false`, `e` is `nil` (a nil `*MyError`), but `return e` wraps it in an `error` interface value with type `*MyError` and value `nil`. That interface value is **not** nil because the type field is populated. `err != nil` is `true`.

Trap: Never return a typed nil pointer in an interface. If you want to return "no error," return the untyped `nil` literal directly:

```
func mayFail(fail bool) error {
    if fail {
        return &MyError{msg: "something went wrong"}
    }
    return nil  // untyped nil: type=nil, value=nil --- this is a nil error
}
```

The fix is to return `nil` (untyped) rather than a variable of the concrete error type.

You can inspect the components of an interface value using reflection, but in practice the fix is always the same: return `nil` directly, not a typed nil pointer.

This trap generalizes beyond pointers. Any **typed nil** boxed in an interface is non-nil — a nil map, a nil slice, a nil function value, or a nil channel all carry a populated type field once stored in an interface, so the interface compares unequal to `nil`.

```
var s []int            // s is a nil slice
var i any = s          // but i is NOT nil: type=[]int, value=nil
fmt.Println(i == nil)  // false
```

The lesson is the same regardless of the underlying kind: to mean "no value," assign or return the untyped `nil` literal, never a typed nil that has been wrapped in an interface.

Trap: Comparing two interface values with `==` compares their dynamic types and then their dynamic values, and it **panics at run time** if the dynamic type is not comparable (a slice, map, or function). This bites hardest when you use interface values as map keys, since inserting a key with a non-comparable dynamic type panics on the spot.

```
var a, b any = []int{1}, []int{1}
fmt.Println(a == b)  // panic: runtime error: comparing uncomparable type []int
```

If you need to compare interface values whose dynamic type might be uncomparable, reach for `reflect.DeepEqual` instead.

Try It

Type this in and run it to watch structural typing, interface composition, a type switch, and `sort.Interface` all working together. Notice that `Track` never says it implements `Playable` — it just has the right methods.

```
package main

import (
    "fmt"
    "sort"
)

// Playable is a tiny interface: anything that can describe itself and report a
// duration satisfies it --- no "implements" needed.
type Playable interface {
    fmt.Stringer
    Seconds() int
}

type Track struct {
    Title  string
```

```
    Artist string
    Length int // seconds
}

func (t Track) String() string { return fmt.Sprintf("%s --- %s", t.Title, t.Artist) }
func (t Track) Seconds() int   { return t.Length }

type byLength []Track

func (b byLength) Len() int           { return len(b) }
func (b byLength) Less(i, j int) bool { return b[i].Length < b[j].Length }
func (b byLength) Swap(i, j int)      { b[i], b[j] = b[j], b[i] }

func describe(p Playable) {
    fmt.Printf("%s (%d s)\n", p, p.Seconds())
}

func main() {
    tracks := []Track{
        {"Espresso", "Sabrina Carpenter", 175},
        {"Good Luck, Babe!", "Chappell Roan", 218},
        {"Birds of a Feather", "Billie Eilish", 210},
    }

    sort.Sort(byLength(tracks))
    for _, t := range tracks {
        describe(t) // Track satisfies Playable structurally
    }

    var v any = tracks[0]
    switch p := v.(type) {
    case Playable:
        fmt.Println("playable:", p)
    default:
        fmt.Println("not playable")
    }
}
```

Try these modifications:

- Add a `Podcast` type with `String()` and `Seconds()` and pass it to `describe` — it satisfies `Playable` for free.
- Replace `sort.Sort(byLength(tracks))` with `slices.SortFunc` and `cmp.Compare` (Chapter 7) to sort by title instead.
- Add a `var _ Playable = Track{}` line at the top of the file, then delete the `Seconds()` method and watch the compile-time check fire.

Key Points

- Go uses **structural typing**: a type satisfies an interface by having the right methods, with no `implements` declaration required.
- Interface composition embeds interfaces inside interfaces; `io.ReadWriter` is `io.Reader` + `io.Writer`.
- `any` (alias for `interface{}`) is the top type; use it sparingly and prefer generics when possible.
- Type assertions come in two forms: `v := i.(T)` panics on failure; `v, ok := i.(T)` is safe.

- Type switches dispatch on dynamic type: `switch v := i.(type) { case int: ... }`.
- The five interfaces to know first: `io.Reader`, `io.Writer`, `fmt.Stringer`, `error`, and `sort.Interface`.
- `sort.Interface` (`Len`, `Less`, `Swap`) lets any type be sorted by `sort.Sort`; prefer `slices.SortFunc` for one-off sorts.
- Accept interfaces, return concrete types — this maximizes flexibility and testability.
- A typed nil pointer stored in an interface is **not** nil; always return the untyped `nil` to signal "no error."

Exercises

1. **Think about it:** Go's structural typing means any package can retroactively make its types satisfy an interface defined in any other package. In Java, if you want your `Song` class to satisfy a new interface `Playable` defined in a library you do not control, you must modify `Song`'s source. Explain how Go's approach changes the relationship between library authors and library users. What does this mean for extending types from packages you cannot modify?

2. **What does this print?**

```go
package main

import "fmt"

type Celsius float64
type Fahrenheit float64

func (c Celsius) String() string {
    return fmt.Sprintf("%.1f°C", float64(c))
}

func printTemp(v fmt.Stringer) {
    fmt.Println(v.String())
}

func main() {
    c := Celsius(37.5)
    f := Fahrenheit(99.5)
    printTemp(c)
    fmt.Println(f)
}
```

3. **Calculation:** An interface value in Go stores two fields: a pointer to type information and a pointer to (or copy of) the data. Given a variable declared as `var r io.Reader = &bytes.Buffer{}`, how many distinct type/value components does `r` hold? If `r` is then assigned `nil`, describe the type and value components of the resulting interface value.

4. **Where is the bug?**

```go
package main

import "fmt"

type DBError struct{ code int }

func (e *DBError) Error() string { return fmt.Sprintf("db error %d", e.code) }

func connect(bad bool) error {
    var err *DBError
```

```
    if bad {
        err = &DBError{code: 500}
    }
    return err
}

func main() {
    e := connect(false)
    if e == nil {
        fmt.Println("connected OK")
    } else {
        fmt.Println("failed:", e)
    }
}
```

5. **Write a program:** Define an interface `Shape` with two methods: `Area() float64` and `Perimeter() float64`. Implement `Shape` for two concrete types: `Rectangle` (with fields `Width` and `Height float64`) and `Circle` (with field `Radius float64`; use `math.Pi`). Write a function `printShapeInfo(s Shape)` that prints the area and perimeter. In `main`, create one `Rectangle` and one `Circle` and call `printShapeInfo` on each.