Gorgo Go for Java Programmers

June 11, 2026

Contents

9 Error Handling **1**
The `error` Interface . . . 1
Creating Errors . . . 2
Wrapping Errors . . . 3
`errors.Is` and `errors.As` . . . 3
`errors.Join` — Combining Multiple Errors . . . 4
Sentinel Errors . . . 6
Custom Error Types . . . 7
`panic` and `recover` . . . 9
The `must` Idiom . . . 10
Go Error-Handling Proverbs . . . 11
Try It . . . 12
Key Points . . . 13
Exercises . . . 13

Chapter 9

Error Handling

Go's approach to errors will feel alien at first if you come from Java — there are no checked exceptions, no `try/catch`, and no exception hierarchy. Instead, errors are plain values that functions return alongside their results, and the caller is expected to inspect them immediately. Java's checked exceptions force the compiler to nag you until every failure path is caught or re-declared; Go makes the opposite bet, trusting you to check the `error` value the function hands back. That trust has teeth: nothing stops you from ignoring a returned error, and when you do, a failed write looks like a successful one, a missing record reads as an empty result, and the bug surfaces three layers away from where it started. Treating errors as explicit values is what keeps Go's control flow readable, but only if you actually look at them. This chapter explains the `error` interface, the conventions around creating and returning errors, how to wrap and unwrap error chains, sentinel errors, custom error types, and the narrow use of `panic/recover`. By the end you will understand why "errors are values" is one of Go's defining proverbs and how to apply it in production code.

The `error` Interface

You saw `error` briefly in Chapter 8 when the standard library interfaces were introduced. Here it gets the full treatment.

`error` is a predeclared interface built into the language itself — not in any package:

```
type error interface {
    Error() string  // returns a human-readable description of the error
}
```

Any type with an `Error() string` method automatically satisfies `error`. That is the entire contract.

Return `error` as the Last Value

The universal Go convention is to return `error` as the **last** return value of a function. On success, return `nil`; on failure, return a non-nil error. Chapter 5 introduced this pattern; here is a refresher:

```
func parseTrackNumber(s string) (int, error) {
    n, err := strconv.Atoi(s)  // convert string to int
    if err != nil {
        return 0, fmt.Errorf("invalid track number %q: %w", s, err)
    }
    return n, nil
}
```

The caller handles it immediately:

```go
n, err := parseTrackNumber("3")
if err != nil {
    log.Fatal(err)
}
fmt.Println("Track:", n)
```

Trap: Java programmers sometimes assign the error to a variable and check it later, or silently ignore it with `_`. In Go, the convention is to check the error **right after the call** and return early. Delaying error checks makes the control flow hard to follow and is not idiomatic. Handle errors first so the success path stays unindented and easy to read. [*error-first-return-early*] and [*no-else-after-error*]

Wut: Go has no checked exceptions. The compiler does not force you to handle an `error` return value. You can write `n, _ := parseTrackNumber(s)` and discard the error entirely — the program will compile. Using `_` is occasionally appropriate (e.g., closing a read-only file), but doing it carelessly is a common source of bugs. `go vet` and linters like `errcheck` can flag unchecked errors. [*no-discard-error*]

Creating Errors

The standard library provides two basic ways to create an error value.

`errors.New`

```go
import "errors"

var ErrEmptyPlaylist = errors.New("playlist is empty")
```

`errors.New` returns an opaque error value whose message is the string you provide. Two calls to `errors.New` with the same string return distinct values — equality is by identity, not by message content.

`fmt.Errorf`

```go
import "fmt"

func loadTrack(id int) (*Track, error) {
    if id <= 0 {
        return nil, fmt.Errorf("loadTrack: invalid id %d", id)
    }
    // ...
    return nil, fmt.Errorf("loadTrack: id %d not found", id)
}
```

`fmt.Errorf` creates an error with a formatted message. It is the workhorse of error creation in Go.

Tip: Prefix error messages with the function or package name: `"loadTrack: ..."`. When errors bubble up through several layers, the prefix chain reads like a stack trace in plain text: `"server: loadTrack: id 42 not found"`. Error strings must be lowercase and must not end with punctuation so they compose cleanly when prefixed. [*lowercase-error-strings*]

Wrapping Errors

Go 1.13 added error wrapping: you can attach a **cause** to an error so that callers can inspect the original error without depending on the message string. The `%w` verb in `fmt.Errorf` is how you wrap:

```
func fetchSong(id int) (*Song, error) {
    row, err := db.LookupSong(id)
    if err != nil {
        return nil, fmt.Errorf("fetchSong %d: %w", id, err)  // wrap the database error
    }
    // ...
}
```

The wrapping error remembers the original `err` and exposes it through `errors.Unwrap`:

```
wrapped := fmt.Errorf("outer: %w", inner)
fmt.Println(errors.Unwrap(wrapped)) // prints inner's message
```

You can wrap multiple levels deep, forming an **error chain**. `errors.Is` and `errors.As` (next section) walk the entire chain, so callers do not need to manually call `errors.Unwrap`.

Tip: Use `%w` whenever you add context to an error and want callers to be able to test for the original cause. Use `%v` (not `%w`) when you want to log the cause for humans but do **not** want callers to programmatically inspect it — for example, when the original error is an implementation detail.

`errors.Is` and `errors.As`

These two functions replace the Java pattern of `catch (SpecificException e)` or `e instanceof SomeException`.

`errors.Is` — Matching Sentinel Values

`errors.Is(err, target)` reports whether any error in `err`'s chain equals `target`.

```
import (
    "errors"
    "io"
)

_, err := reader.Read(buf)
if errors.Is(err, io.EOF) {
    fmt.Println("reached end of stream")
}
```

Trap: Never compare errors with `==` when they might be wrapped. `wrappedErr == io.EOF` is `false` if `wrappedErr` was produced by `fmt.Errorf("read: %w", io.EOF)`. `errors.Is(wrappedErr, io.EOF)` correctly walks the chain and returns `true`.

`errors.As` — Extracting a Concrete Type

`errors.As(err, &target)` walks the chain and, if it finds an error assignable to `*target`, sets `target` to that error and returns `true`. This is Go's replacement for `instanceof` with a cast.

```go
type TrackError struct {
    TrackID int
    Reason  string
}

func (e *TrackError) Error() string {
    return fmt.Sprintf("track %d: %s", e.TrackID, e.Reason)
}

func process(err error) {
    var te *TrackError
    if errors.As(err, &te) {
        fmt.Printf("problem with track %d: %s\n", te.TrackID, te.Reason)
    }
}
```

In Java you would write:

```java
if (err instanceof TrackException te) {
    System.out.println("problem with track " + te.trackId);
}
```

The Go form is slightly more verbose but it works through any number of wrapping layers.

Tip: Use `errors.Is` when you want to test for a **specific value** (a sentinel like `io.EOF`). Use `errors.As` when you want to test for a **specific type** and access its fields.

`errors.Join` — Combining Multiple Errors

Go 1.20 added `errors.Join`, which combines multiple errors into one. The combined error's `Error()` method returns each non-nil message on its own line, and `errors.Is`/`errors.As` check all of them.

```go
import "errors"

func validatePlaylist(name, owner string, trackCount int) error {
    var errs []error

    if name == "" {
        errs = append(errs, errors.New("name is required"))
    }
    if owner == "" {
        errs = append(errs, errors.New("owner is required"))
    }
    if trackCount < 0 {
        errs = append(errs, errors.New("track count cannot be negative"))
    }

    return errors.Join(errs...)  // nil if errs is empty
}
```

`errors.Join(nil, nil)` returns `nil`, so you can always pass the slice with `...` and get a nil result when there are no errors.

Tip: `errors.Join` is the idiomatic tool for **validation** patterns where you want to report all problems at once rather than stopping at the first one. In Java you might collect exceptions in a list and throw a composite exception at the end; `errors.Join` is the Go equivalent.

A practical example with Andrew Spencer's discography validator:

```
package main

import (
    "errors"
    "fmt"
)

type Album struct {
    Title    string
    Artist   string
    Year     int
    TrackCnt int
}

func validateAlbum(a Album) error {
    var errs []error
    if a.Title == "" {
        errs = append(errs, errors.New("title is required"))
    }
    if a.Artist == "" {
        errs = append(errs, errors.New("artist is required"))
    }
    if a.Year < 1900 || a.Year > 2100 {
        errs = append(errs, fmt.Errorf("year %d is out of range", a.Year))
    }
    if a.TrackCnt <= 0 {
        errs = append(errs, errors.New("track count must be positive"))
    }
    return errors.Join(errs...)
}

func main() {
    a := Album{
        Title:    "Zombie",
        Artist:   "Andrew Spencer",
        Year:     2022,
        TrackCnt: 23,
    }
    if err := validateAlbum(a); err != nil {
        fmt.Println("invalid:", err)
    } else {
        fmt.Println("album OK:", a.Title)
    }

    bad := Album{Title: "", Artist: "", Year: 1800, TrackCnt: -1}
    if err := validateAlbum(bad); err != nil {
        fmt.Println("invalid album:")
        fmt.Println(err)
```

```
    }
}
```

Output:

```
album OK: Zombie
invalid album:
title is required
artist is required
year 1800 is out of range
track count must be positive
```

Sentinel Errors

A **sentinel error** is a package-level error variable used as a well-known signal. The caller compares against it with `errors.Is`.

The most famous sentinel in Go is `io.EOF`:

```
// in package io
var EOF = errors.New("EOF")
```

`io.EOF` signals that there is no more data to read. It is not a failure — it is expected behavior at the end of a stream. Functions like `bufio.Scanner.Scan` and `io.Copy` consume it internally as a termination signal — a successful `io.Copy` returns `nil`, not `io.EOF`.

Other common sentinels you will encounter:

```
// database/sql
var ErrNoRows = errors.New("sql: no rows in result set")

// os (aliases of the io/fs sentinels, tested via errors.Is)
var ErrNotExist = fs.ErrNotExist     // "file does not exist"
var ErrPermission = fs.ErrPermission // "permission denied"
```

Tip: By convention, sentinel error variable names start with `Err`. Define your own sentinels as package-level `var` declarations using `errors.New`. Export them (capitalize them) when callers outside your package need to test for them.

Trap: Do not test sentinel errors with `==`. Wrap-aware callers must use `errors.Is`:

```
// Wrong: misses wrapped errors
if err == io.EOF { ... }

// Right: walks the entire chain
if errors.Is(err, io.EOF) { ... }
```

Here is a realistic read loop using `io.EOF` as a sentinel:

```
package main

import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "strings"
)
```

```
func main() {
    // Robert Dream House, Darude, Chicane --- trance classics
    src := strings.NewReader("Children\nSandstorm\nSaltwater\n")
    r := bufio.NewReader(src)

    for {
        line, err := r.ReadString('\n')
        if len(line) > 0 {
            fmt.Print("track: ", line)
        }
        if errors.Is(err, io.EOF) {
            break
        }
        if err != nil {
            fmt.Println("unexpected error:", err)
            break
        }
    }
}
```

Output:

```
track: Children
track: Sandstorm
track: Saltwater
```

Custom Error Types

When an error needs to carry structured data — a status code, a field name, a record ID — define a custom error type. Implement the `error` interface by adding an `Error() string` method.

```
// StreamError is returned when a streaming request fails.
type StreamError struct {
    TrackID  string  // ID of the track that failed
    HTTPCode int     // HTTP status code from the streaming service
    Msg      string  // human-readable reason
}

func (e *StreamError) Error() string {
    return fmt.Sprintf("stream error %d for track %q: %s", e.HTTPCode, e.TrackID, e.Msg)
}
```

Return it as the `error` interface:

```
func streamTrack(id string) error {
    if id == "" {
        return &StreamError{TrackID: id, HTTPCode: 400, Msg: "empty track ID"}
    }
    // ... call streaming API ...
    return nil
}
```

The caller uses `errors.As` to recover the structured data:

```
err := streamTrack("")
var se *StreamError
```

```
if errors.As(err, &se) {
    fmt.Printf("HTTP %d: %s\n", se.HTTPCode, se.Msg)
}
// HTTP 400: empty track ID
```

Tip: `errors.As` matches the **concrete type** stored in the error chain — it has nothing to do with receiver style by itself. The receiver you choose decides *which* concrete type satisfies `error`: a pointer receiver makes `*StreamError` the error type, while a value receiver makes `StreamError` the error type. Whatever you return is what ends up in the chain, and the `target` you pass to `errors.As` must point to that same type (`var se *StreamError` paired with returning `&StreamError{...}`). The standard practice for custom error types is a pointer receiver plus returning `*StreamError` values; mixing a value receiver with `var se *StreamError` is the classic mismatch that makes `errors.As` quietly fail to match.

Adding Context with `Unwrap`

If your custom error wraps another error, implement `Unwrap() error` so that `errors.Is` and `errors.As` can traverse your type:

```
type FetchError struct {
    URL string  // URL that was fetched
    Err error   // the underlying error
}

func (e *FetchError) Error() string {
    return fmt.Sprintf("fetch %s: %s", e.URL, e.Err)
}

func (e *FetchError) Unwrap() error {
    return e.Err  // expose the wrapped error for errors.Is / errors.As
}
```

Now `errors.Is(fetchErr, io.EOF)` will correctly check whether the underlying error is `io.EOF`, even though the outer type is `*FetchError`.

A fuller example using DJ Analyzer's "Insomnia":

```
package main

import (
    "errors"
    "fmt"
    "io"
)

// PlaybackError wraps an underlying error with playback context.
type PlaybackError struct {
    Track string  // track that failed to play
    Err   error   // underlying cause
}

func (e *PlaybackError) Error() string {
    return fmt.Sprintf("playback failed for %q: %s", e.Track, e.Err)
}
```

```go
func (e *PlaybackError) Unwrap() error {
    return e.Err  // allows errors.Is / errors.As to walk the chain
}

func play(track string) error {
    // simulate an EOF from a truncated audio stream
    return &PlaybackError{Track: track, Err: io.EOF}
}

func main() {
    err := play("Insomnia")
    fmt.Println(err)

    if errors.Is(err, io.EOF) {
        fmt.Println("stream ended unexpectedly --- retry or skip")
    }

    var pe *PlaybackError
    if errors.As(err, &pe) {
        fmt.Println("affected track:", pe.Track)
    }
}
```

Output:

```
playback failed for "Insomnia": EOF
stream ended unexpectedly --- retry or skip
affected track: Insomnia
```

`panic` and `recover`

Go has `panic` and `recover`, but they are **not** the normal error-handling mechanism. Java programmers sometimes reach for `panic` as a `RuntimeException` substitute — that is the wrong model.

`panic`

`panic` stops the normal execution of the current goroutine, unwinds the call stack running any deferred functions, and terminates the program with a message (unless a deferred function recovers, as described below).

```go
func mustPositive(n int) int {
    if n <= 0 {
        panic(fmt.Sprintf("mustPositive: got %d, want > 0", n))
    }
    return n
}
```

Use `panic` only for **truly unrecoverable situations**:

- A programming error that should have been caught at compile time (e.g., misused internal API).
- Initialization failures so severe the program cannot run at all (e.g., a required config file is missing at startup).
- Invariant violations that indicate a bug, not a runtime condition.

Never use `panic` for expected failure conditions like "file not found" or "invalid user input." Return an `error` instead.

Trap: A common Java habit is to use unchecked exceptions for "should never happen" cases and let them propagate freely. In Go, `panic` is the analog, but the bar is much higher. If callers can reasonably be expected to handle the failure, return an `error`. `panic` is a last resort, not a shortcut. [*errors-not-panic*]

`recover`

`recover` catches a panic in a deferred function and returns the value that was passed to `panic`. Outside of a deferred function, `recover` returns `nil` and has no effect.

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

    // pretend this panics on a bad track
    if track == "" {
        panic("empty track name")
    }
    fmt.Println("playing:", track)
    return nil
}

err := safePlay("")
fmt.Println(err)
// safePlay: recovered panic: empty track name

err = safePlay("Sandstorm")
// playing: Sandstorm
```

Tip: The primary legitimate use of `recover` is inside a **library boundary** — converting an unexpected internal panic into an error that the library returns to its caller. This prevents library bugs from crashing the entire program. The standard `encoding/json` package uses this pattern internally. Do not use `recover` as a general exception handler the way you might use a `catch (Exception e)` block in Java.

Wut: `panic` and `recover` are not the same as Java exceptions. Java exceptions propagate up the call stack and can be caught at any frame with a `try/catch`. Go's `recover` only works in **deferred functions** in the **same goroutine** that panicked. You cannot `recover` a panic from a different goroutine.

The `must` Idiom

Some initialization errors are programmer mistakes, not runtime conditions — a bad regex pattern or a malformed template literal is a bug, not something you should silently swallow or handle per-request. For these cases the standard library provides `Must` wrappers that panic on error:

```
// regexp.MustCompile compiles a pattern and panics if the syntax is invalid.
var trackTitleRE = regexp.MustCompile(`^[A-Za-z0-9 ,'!?&()\-]+$`)

func validTitle(s string) bool {
```

```
    return trackTitleRE.MatchString(s)
}
```

The standard library ships only a handful of these wrappers, one bolted onto each package that happened to need it: `regexp.MustCompile`, `template.Must`, `netip.MustParseAddr`, and a few others. They all do the same thing — take a `(value, error)` pair, panic if the error is non-nil, and otherwise return the value — so writing a new one for every type that returns `(T, error)` is tedious. With generics (Chapter 18) you can write that pattern **once** and reuse it for any such function:

```
// Must unwraps a (value, error) pair, panicking if err is non-nil.
func Must[T any](v T, err error) T {
    if err != nil {
        panic(err)
    }
    return v
}
```

Because `Must` returns the unwrapped value directly, it drops into any package-level `var` that would otherwise need a dedicated `Must` wrapper — no matter the type:

```
var trackTmpl = Must(template.New("track").Parse(
    `{{.Title}} by {{.Artist}} ({{.BPM}} BPM)`,
))

var nyc = Must(time.LoadLocation("America/New_York")) // *time.Location or panic
```

These patterns are appropriate because the panic fires at program startup (package-level `var` blocks and `init()` run before `main`), so a bug in the literal is caught immediately rather than surfacing as a mysterious runtime failure on the first request.

Tip: This little generic `Must` is one of the most-copied helpers in Go; many teams keep a one-line `must` package for it. It exists because Go 1.18 added generics *after* the standard library had already hand-written `MustCompile`, `template.Must`, and the rest — a single generic function now subsumes them all.

Trap: Never call `regexp.MustCompile` or `Must` inside a function that runs per-request or in a loop. They are for compile-time-constant literals only. If the pattern or template text comes from user input or configuration, use `regexp.Compile` or `template.New(...).Parse(...)` and handle the error normally.

Go Error-Handling Proverbs

Two of Go's most quoted proverbs apply directly to this chapter.

"Errors are values"

Rob Pike's essay of the same name (Pike 2015) makes a key point: because errors are just values, you can write functions that operate on them, store them in structs, and compose them with the same tools you use for any other value. This is why `errors.Join`, custom error types, and wrapping all feel natural in Go. Java's exceptions, by contrast, are objects thrown out of the normal return path — they are harder to treat as first-class data.

"Don't just check errors, handle them gracefully"

Checking `if err != nil` and immediately calling `log.Fatal` is not handling an error — it is panicking politely. Handling an error means deciding what to do: retry, return a friendlier message, fall back to a default, or accumulate it with `errors.Join` and report all problems at once. The goal is code that is predictable and informative when things go wrong.

Try It

Type this in and run it. It pulls together the chapter's main moves — a sentinel error, a custom error type with `Unwrap`, and the `errors.Is`/`errors.As` pair — in one small lookup. Watch how the same returned `error` answers two different questions: "is this *kind* of failure?" (`errors.Is`) and "what *value* failed?" (`errors.As`).

```
package main

import (
    "errors"
    "fmt"
)

var ErrTrackNotFound = errors.New("track not found")

// LookupError carries the requested title alongside the underlying cause.
type LookupError struct {
    Title string  // title the caller asked for
    Err   error   // underlying cause (often ErrTrackNotFound)
}

func (e *LookupError) Error() string {
    return fmt.Sprintf("lookup %q: %s", e.Title, e.Err)
}

func (e *LookupError) Unwrap() error {
    return e.Err  // lets errors.Is / errors.As walk the chain
}

func findBPM(title string) (int, error) {
    catalog := map[string]int{
        "Quédate":   108,  // Bizarrap & Quevedo, BZRP Music Sessions #52
        "Sandstorm": 136,
    }
    bpm, ok := catalog[title]
    if !ok {
        return 0, &LookupError{Title: title, Err: ErrTrackNotFound}
    }
    return bpm, nil
}

func main() {
    for _, title := range []string{"Quédate", "Inexistente"} {
        bpm, err := findBPM(title)
        if errors.Is(err, ErrTrackNotFound) {
            var le *LookupError
            if errors.As(err, &le) {
```

```
                fmt.Printf("no encontrado: %q (%v)\n", le.Title, err)
            }
            continue
        }
        if err != nil {
            fmt.Println("unexpected:", err)
            continue
        }
        fmt.Printf("%s is %d BPM\n", title, bpm)
    }
}
```

Then try these modifications:

- Switch the `Error()` and `Unwrap()` methods to value receivers and change `var le *LookupError` to `var le LookupError`; confirm `errors.As` still matches when you return `LookupError{...}` (no `&`), then deliberately mismatch them and watch it fail.
- Wrap the returned error one more level with `fmt.Errorf("findBPM: %w", err)` and confirm both `errors.Is` and `errors.As` still see through the extra layer.
- Replace `%w` with `%v` in that extra wrap and observe how `errors.Is` now returns `false`.

Key Points

- `error` is a predeclared interface: `Error() string`; any type with that method satisfies it.
- Return `error` as the last value; return `nil` for success; check the error immediately after the call.
- `errors.New` creates a simple error; `fmt.Errorf` creates a formatted one.
- `fmt.Errorf` with `%w` wraps an error, forming a chain; use `%v` when you do not want callers to inspect the cause.
- `errors.Is(err, target)` walks the chain to match a sentinel value; never use `==` for this.
- `errors.As(err, &target)` walks the chain to match a concrete type; it is Go's replacement for `instanceof` + cast.
- `errors.Join` (Go 1.20) combines multiple errors; ideal for validation that should report all failures at once.
- Sentinel errors (e.g., `io.EOF`, `sql.ErrNoRows`) are package-level `var` values; name them `ErrFoo`.
- Custom error types implement `error` with a pointer receiver; add `Unwrap() error` if they wrap another error.
- `panic` is for unrecoverable programming errors, not for expected runtime conditions — return an `error` instead.
- `recover` only works in deferred functions in the same goroutine; its main use is converting internal panics into errors at a library boundary.
- "Errors are values": handle them with the same composability you apply to any other data.

Exercises

1. **Think about it:** Java uses checked exceptions to force callers to handle failures. Go returns `error` values that the compiler does not require you to inspect. What are the trade-offs of each approach? In what situations does Go's approach lead to more reliable code, and in what situations might it lead to less reliable code compared to Java's checked exceptions?

2. **What does this print?**

```
package main

import (
```

```
    "errors"
    "fmt"
)

var ErrNotFound = errors.New("not found")

type CatalogError struct {
    Track string
    Err   error
}

func (e *CatalogError) Error() string {
    return fmt.Sprintf("catalog: %s: %s", e.Track, e.Err)
}

func (e *CatalogError) Unwrap() error {
    return e.Err
}

func lookup(track string) error {
    return &CatalogError{Track: track, Err: ErrNotFound}
}

func main() {
    err := lookup("Insomnia")
    fmt.Println(err)
    fmt.Println(errors.Is(err, ErrNotFound))

    var ce *CatalogError
    if errors.As(err, &ce) {
        fmt.Println(ce.Track)
    }
}
```

3. **Calculation:** Consider the following code. For the input `Song{Title: "", Artist: "DJ Analyzer", Year: 2021, BPM: -1}`, how many sub-errors does the joined error returned by `validateSong` contain? What is the output of `fmt.Println(err)` for that input?

```
package main

import (
    "errors"
    "fmt"
)

type Song struct {
    Title  string
    Artist string
    Year   int
    BPM    int
}

func validateSong(s Song) error {
    var errs []error
    if s.Title == "" {
```

```
        errs = append(errs, errors.New("title required"))
    }
    if s.Year < 2000 || s.Year > 2030 {
        errs = append(errs, fmt.Errorf("year %d out of range", s.Year))
    }
    if s.BPM <= 0 {
        errs = append(errs, errors.New("BPM must be positive"))
    }
    return errors.Join(errs...)
}

func main() {
    s := Song{Title: "", Artist: "DJ Analyzer", Year: 2021, BPM: -1}
    err := validateSong(s)
    fmt.Println(err)
}
```

4. **Where is the bug?**

```
package main

import (
    "fmt"
    "io"
    "strings"
)

func readAll(r io.Reader) ([]byte, error) {
    buf := make([]byte, 4)
    var result []byte
    for {
        n, err := r.Read(buf)
        result = append(result, buf[:n]...)
        if err == io.EOF {
            break
        }
        if err != nil {
            return nil, fmt.Errorf("readAll: %w", err)
        }
    }
    return result, nil
}

func main() {
    r := strings.NewReader("Children")
    data, err := readAll(r)
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println(string(data))
}
```

5. **Write a program:** Write a function `parseTimecode(s string) (int, int, error)` that parses a string in the format `"MM:SS"` (e.g., `"03:45"`) and returns the minutes, seconds, and `nil`. Return a descriptive

error using `fmt.Errorf` if the string is not in the expected format, if either part is not a valid integer, or if minutes or seconds are out of range (minutes >= 0, seconds 0–59). Define a sentinel `var ErrInvalidTimecode = errors.New("invalid timecode")` and wrap it with `%w` in your error returns so that callers can use `errors.Is(err, ErrInvalidTimecode)`. In `main`, call `parseTimecode` with at least three inputs: one valid (`"03:45"`), one with a bad format (`"345"`), and one with an out-of-range second (`"01:61"`). Print the result or error for each.

Pike, Rob. 2015. "Errors are values." The Go Blog. https://go.dev/blog/errors-are-values.