Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 6

Objects using Methods and Embedding

Java bundles data, behavior, and code reuse together in a single class construct. Go separates them into three distinct mechanisms: structs hold data, methods attach behavior to structs, and embedding provides composition-based code reuse. This chapter covers all three, explains how to write constructors and handle resource cleanup without the language features Java provides for those tasks, and shows why Go's "no inheritance" stance is a strength rather than a limitation.

Receivers and Methods

Like Java, Go lets you attach methods to a type, but it declares them quite differently. Consider the following class:

```
class Point {
    int x;
    int y;
    void scale(int factor) { x *= factor; y *= factor; }
}
```

In Java you define the member variables of a class in the same place you define all the methods of a class. Go does it differently. You define the member variables with `struct`, and the methods are declared later to operate on that structure using *receivers* that have the following forms:

```
func (t T) MethodName(params) returnType    // value receiver --- gets a copy of T
func (t *T) MethodName(params) returnType   // pointer receiver --- gets a pointer to T
```

Method declarations look just like normal function declarations except for the receiver variables, shown as `t` in the above examples. These receiver variables act like the `this` pointer in Java with two big differences. First, you can pick the name. Go best practices say that it should be a one- or two-letter abbreviation of the type name. [***receiver-name-abbreviation***] Second, a receiver can act on a pointer to an object — like `this` in Java — or it can act on a copy of the object. The former is called a pointer receiver, the latter a value receiver.

Using `Point` again as a minimal example:

```
type Point struct {
    X, Y int
}

// Value receiver --- Scale gets a copy of the Point.
func (p Point) ScaleBad(factor int) {
    p.X *= factor   // modifies the copy only
    p.Y *= factor
```

```
}

// Pointer receiver --- Scale gets a pointer to the original Point.
func (p *Point) Scale(factor int) {
    p.X *= factor   // modifies the original
    p.Y *= factor
}

pt := Point{X: 3, Y: 4}

pt.ScaleBad(10)
fmt.Println(pt) // {3 4} --- unchanged; ScaleBad worked on a copy

pt.Scale(10)
fmt.Println(pt) // {30 40} --- changed; Scale worked on the original
```

Go is convenient here: even though `pt` is a plain `Point` value (not a `*Point`), you can call `pt.Scale(10)` and Go automatically takes the address for you — it is equivalent to `(&pt).Scale(10)`.

Trap: The `Point` example above deliberately mixes a value receiver (`ScaleBad`) with a pointer receiver (`Scale`) to show the behavioral difference. In real code, `Point` would have `Scale` as its only method (or all methods would use `*Point`). The name `ScaleBad` is the hint: drop the value-receiver `ScaleBad` and use only pointer receivers.

Tip: If any method on a type needs a pointer receiver, make all methods on that type pointer receivers. Mixing value and pointer receivers on the same type confuses the method set rules and makes interface satisfaction harder to reason about. This is a firm Go idiom worth following from day one. [***no-mixed-receivers***]

Wut: If you call a pointer receiver method through a value that is not addressable — for example, the return value of a function call used directly — Go cannot take the address and the compiler reports an error. Chapter 2 introduced & and pointers; a value is addressable when you could legally apply & to it, which is why a bare function return value does not qualify.

The same rules apply to any type. Here is a `Track` type with two pointer-receiver methods:

```
func (t *Track) String() string {              // pointer receiver: consistent with ScaleBPM
    return fmt.Sprintf("%s by %s (%d BPM)", t.Title, t.Artist, t.BPM)
}

func (t *Track) ScaleBPM(factor float64) {     // pointer receiver: mutates BPM
    t.BPM = int(float64(t.BPM) * factor)
}
```

`ScaleBPM` mutates `BPM`, so it needs `*Track`. [***pointer-receiver-for-mutation***] `String` uses `*Track` too, even though it only reads, to keep receivers consistent.

Unlike Java, where all methods of a class must live inside that class's file, Go methods can be declared in any file within the same package — which means the same directory. A single file can also define structs and methods for several different types. There is no rule that says `track.go` must contain only `Track`-related code. The constraint is the package boundary: a type and its methods must all be in the same package, but they can be spread across as many files as you like.

How you split code across files is a style question. The Google Go Style Guide recommends keeping files focused and reasonably short: "files should be focused enough that a maintainer can tell which file contains

something, and the files should be small enough that it will be easy to find once there" (Google 2024). A common pattern is one file per major type, with its closely related helpers alongside it.

Method Sets

Every type has a **method set** — the set of methods you can call on a value of that type.

Value type	Methods callable
`T`	only value-receiver methods of `T`
`*T`	all methods of `T` (both value and pointer receivers)

A `*T` pointer can call everything: pointer-receiver methods (directly) and value-receiver methods (Go automatically dereferences). A plain `T` value can only call value-receiver methods.

Wut: You might expect that `T` can call pointer-receiver methods too, because Go auto-takes addresses for you when you write `t.ScaleBPM(1.1)` on an addressable variable. It can — but only for *calls*, not for interface satisfaction. When you store a `T` (not `*T`) in an interface, only the value-receiver methods are in the method set. This distinction matters in Chapter 8.

Calling Methods

Go is flexible about how you call methods on values vs pointers:

```
t := Track{Title: "Flaming June", Artist: "BT", BPM: 118}

// Calling a pointer-receiver method on an addressable value --- Go auto-takes address.
fmt.Println(t.String())    // equivalent to (&t).String() --- Flaming June by BT (118 BPM)
t.ScaleBPM(1.1)            // equivalent to (&t).ScaleBPM(1.1)
fmt.Println(t.BPM)         // 129

// Calling a pointer-receiver method on a pointer --- straightforward.
p := &t
fmt.Println(p.String())    // Flaming June by BT (129 BPM)
```

Go's auto-dereference and auto-address rules mean you rarely write `(&t).Method()` or `(*p).Method()` yourself.

Attaching Methods to the Playlist

Using the `Track` type from Chapter 2 plus a new `Playlist` type, here is a complete example of both receiver kinds in one type:

```
type Track struct {
    Title    string
    Artist   string
    BPM      int
    Duration float64 // seconds
}

type Playlist struct {
    Name   string
    tracks []Track
}
```

```go
func (p *Playlist) Add(t Track) {         // pointer receiver: mutates the slice
    p.tracks = append(p.tracks, t)
}

func (p *Playlist) Len() int {            // pointer receiver: consistent with Add
    return len(p.tracks)
}

func (p *Playlist) AverageBPM() float64 { // pointer receiver: consistent with Add
    if len(p.tracks) == 0 {
        return 0
    }
    total := 0
    for _, t := range p.tracks {
        total += t.BPM
    }
    return float64(total) / float64(len(p.tracks))
}

pl := Playlist{Name: "Late Night Vibes"}
pl.Add(Track{Title: "Flaming June", Artist: "BT", BPM: 118})
pl.Add(Track{Title: "Emerald Triangle 2012", Artist: "Angoscia", BPM: 127})
pl.Add(Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 97})
fmt.Println(pl.Len())                      // 3
fmt.Printf("%.1f BPM\n", pl.AverageBPM())  // 114.0 BPM
```

Tip: All three methods use pointer receivers because `Add` needs one, and the rule is: if any method on a type needs a pointer receiver, make them all pointer receivers. `tracks` is also unexported (lowercase), so callers outside this package can only grow the playlist through `Add`, which lets you add validation later without breaking the public API.

Methods on Non-Struct Types

This is where Go and Java part ways. In Java, methods can only live inside a class. A bare `int` or `String` can never grow a method of your own. Go has no such rule: you can attach a method to *any* named type, not just a struct. A named numeric type, a named string type, even a named slice or map type can have methods.

The only requirement is that you define the named type yourself, in the same package as the method. Here is a `BPM` type whose underlying type is `int`, with a method that classifies the tempo:

```go
type BPM int

// Value receiver --- Genre just reads the number.
func (b BPM) Genre() string {
    switch {
    case b < 100:
        return "downtempo"
    case b < 125:
        return "house"
    default:
        return "trance"
    }
}
```

```
b := BPM(118)
fmt.Println(b.Genre()) // house
fmt.Println(b + 10)    // 128 --- still behaves like an int
```

`BPM` keeps all the arithmetic behavior of its underlying `int` — you can add, compare, and print it like a number — but it also carries the `Genre` method you defined. A value receiver is fine here because `Genre` only reads `b`; reach for a pointer receiver only when a method mutates the value.

Named slice and map types work the same way, and this is how you give a collection its own behavior without wrapping it in a struct:

```
type Tracklist []Track

func (tl Tracklist) TotalSeconds() float64 {
    var total float64
    for _, t := range tl {
        total += t.Duration
    }
    return total
}
```

Even a *function* type can have methods. This sounds exotic, but it is the trick behind one of Go's most common idioms: adapting a plain function so it satisfies an interface — the **adapter** or **strategy** pattern from Java, with no class boilerplate. Define a named function type, give it the method the interface requires, and the method body simply calls the function value (its own receiver):

```
type TrackFilter func(Track) bool

// Negate returns a filter that accepts exactly the tracks f rejects.
func (f TrackFilter) Negate() TrackFilter {
    return func(t Track) bool { return !f(t) }
}

var isLong TrackFilter = func(t Track) bool { return t.Duration > 300 }
isShort := isLong.Negate()
fmt.Println(isShort(Track{Duration: 120})) // true
```

The standard library uses exactly this pattern: `http.HandlerFunc` is a named function type with a `ServeHTTP` method, which lets an ordinary function stand in wherever an `http.Handler` interface value is expected (Chapter 15).

Trap: You can only attach a method to a type that is defined in the current package. Writing `func (i int) Double() int` is a compile error (*cannot define new methods on non-local type int*), and so is attaching a method directly to `time.Duration` or any other type from another package. The fix is the same in both cases: define your own named type (`type BPM int`) and hang the method on that.

Constructors

Go has **no constructor syntax**. There is no `new SomeClass(...)` keyword, no `__init__` method, and no special function that runs automatically when a struct is created.

A zero-value struct is valid on its own — that is by design. When you need a struct that starts in a specific non-zero state, or when you want to validate inputs at creation time, the idiomatic Go replacement is a **`New*` factory function**.

The New* Pattern

By convention, a factory function is named `New` followed by the type name. `go doc` and IDE tools recognize this pattern and display it alongside the type.

```
func NewPlaylist(name string) *Playlist {
    return &Playlist{Name: name}
}
```

The return type is `*Playlist` rather than `Playlist` so the caller gets a pointer and can immediately call pointer-receiver methods like `Add`. [*return-concrete-types*]

```
pl := NewPlaylist("Late Night Vibes")
pl.Add(Track{Title: "Flaming June", Artist: "BT", BPM: 118})
```

Compare with Java:

```
// Java
Playlist pl = new Playlist("Late Night Vibes");

// Go
pl := NewPlaylist("Late Night Vibes")
```

The call sites look nearly identical. The Go version is a plain function call with no special language support — but that is all you need.

Constructors That Validate

Factory functions can return an `error` when the inputs are invalid. A Java constructor throws an exception; a Go factory function returns the error as a second value. Error strings should be lowercase and not end with punctuation [*lowercase-error-strings*], as the `NewTrack` example below demonstrates.

```
func NewTrack(title, artist string, bpm int) (Track, error) {
    if title == "" {
        return Track{}, fmt.Errorf("newTrack: title must not be empty")
    }
    if artist == "" {
        return Track{}, fmt.Errorf("newTrack: artist must not be empty")
    }
    if bpm <= 0 || bpm > 300 {
        return Track{}, fmt.Errorf("newTrack: BPM %d is out of range [1, 300]", bpm)
    }
    return Track{Title: title, Artist: artist, BPM: bpm}, nil
}

t, err := NewTrack("Emerald Triangle 2012", "Angoscia", 127)
if err != nil {
    log.Fatal(err)
}
fmt.Println(t.Title) // Emerald Triangle 2012
```

Tip: Return `(T, error)` — a value, not a pointer — when the zero value of `T` is harmless and `T` is small. Return `(*T, error)` when the caller needs to call pointer-receiver methods immediately after construction, or when `T` is large enough that you want to avoid copying it.

Wut: In Java, throwing in a constructor is the only way to signal a construction failure. In Go, a factory function returns `(nil, err)` or `(T{}, err)`. There is no special "failed construction" state — the error is just a value you check like any other.

Destructors

Go has **no destructor syntax**. There is no `finalize()`, no `__del__`, and no `~ClassName()`. The garbage collector reclaims heap memory automatically; you do not manage object lifetimes.

For resources that need explicit cleanup — open files, network connections, mutex locks, database transactions — Go's answer is `defer`.

defer for Cleanup

`defer` was introduced in Chapter 4. The key rule: a deferred call runs when the surrounding function returns, in LIFO order. This makes it the idiomatic replacement for Java's try-with-resources.

The standard open/close pattern:

```
f, err := os.Open(path)
if err != nil {
    return err
}
defer f.Close()  // runs when the enclosing function returns, no matter how
// ... use f ...
```

The `defer f.Close()` line is written immediately after the successful open, before any logic that might return early on error. That placement ensures `f.Close()` is called on every path out of the function.

Compare the patterns side by side:

```
// Java: try-with-resources
try (FileReader f = new FileReader(path)) {
    // use f --- close is called automatically on exit
} catch (IOException e) {
    // handle error
}
```

```
// Go: defer
f, err := os.Open(path)
if err != nil {
    return err
}
defer f.Close()
// use f --- Close is called automatically on return
```

Both guarantee that the resource is released even if the body returns early. The Go form scales naturally to multiple resources:

```
db, err := sql.Open("postgres", dsn)
if err != nil {
    return err
}
defer db.Close()

tx, err := db.Begin()
if err != nil {
```

```
    return err
}
defer tx.Rollback() // rolls back only if Commit has not been called
```

Trap: The arguments to a deferred call — including the receiver — are evaluated at the moment the `defer` statement executes, not when the deferred function actually runs. Wrapping the call in a closure defers the evaluation of any captured variables until the function returns, which matters when you reassign one of those variables afterward:

```
f, _ := os.Open("a.txt")
defer f.Close()          // captures THIS f right now

f, _ = os.Open("b.txt")   // reassigns f
// the deferred call still closes a.txt, not b.txt
```

The closure form reads the variable at return time instead:

```
f, _ := os.Open("a.txt")
defer func() { f.Close() }() // reads f when main returns

f, _ = os.Open("b.txt")      // reassigns f
// now the deferred call closes b.txt
```

Wut: The Go documentation writes `defer f.Close()` everywhere, yet that very pattern violates the *Errors are values* proverb — it throws away the error that `Close` returns. For a file you only read from, ignoring it is usually fine: you already have the bytes you wanted. But some errors surface *only* on `Close`. A buffered writer can hold data that is not flushed until `Close` runs, and even for a plain `os.File` some filesystems report a failed write only when the file is closed — `defer f.Close()` would silently swallow it. When the close can lose data, capture the error instead of deferring it blindly: assign it to a named return value from a deferred closure, or call `Close` explicitly on the happy path.

```
func save(path string, data []byte) (err error) {
    f, err := os.Create(path)
    if err != nil {
        return err
    }
    defer func() {
        if cerr := f.Close(); cerr != nil && err == nil {
            err = cerr // surface a close failure the caller would miss
        }
    }()
    _, err = f.Write(data)
    return err
}
```

runtime.SetFinalizer

The `runtime` package has `SetFinalizer(obj, finalizer)`, which registers a function to run when the GC is about to collect `obj`. It exists for interoperability with C libraries that require explicit deallocation, and for similar low-level needs.

In almost every other situation, `SetFinalizer` is the wrong tool. Finalizers run at an unpredictable time, may never run at all if the program exits normally, and interact poorly with GC tuning.

Trap: Do not use `runtime.SetFinalizer` to release file handles, locks, or network connections. Use `defer` instead — it is deterministic, runs immediately on function exit, and is far easier to reason about.

Embedding

Go has no inheritance. The mechanism for code reuse between types is **embedding**: you include one struct inside another by naming only the type, without a field name.

The fields and methods of the embedded type are **promoted** to the outer type — you can access them directly as if they were declared on the outer type itself.

Field and Method Promotion

```
type Artist struct {
    Name    string
    Country string
}

func (a Artist) Label() string {
    return fmt.Sprintf("%s (%s)", a.Name, a.Country)
}

type Song struct {
    Artist        // embedded --- no field name
    Title  string
    BPM    int
}

s := Song{
    Artist: Artist{Name: "Angoscia", Country: "Italy"},
    Title:  "Emerald Triangle 2012",
    BPM:    127,
}

fmt.Println(s.Name)    // Angoscia  --- promoted from Artist
fmt.Println(s.Country) // Italy    --- promoted from Artist
fmt.Println(s.Label()) // Angoscia (Italy) --- promoted method
fmt.Println(s.Title)   // Emerald Triangle 2012 --- own field
```

You can still reach the embedded struct directly by its type name when you need to:

```
fmt.Println(s.Artist.Name) // Angoscia --- explicit path
```

The explicit path is also how you distinguish between an outer field and an embedded field when there is a name collision — covered below.

Embedded Value vs Embedded Pointer

You can embed either a value or a pointer to a type:

```
type Song struct {
    Artist        // embedded value --- Song owns the Artist data
    Title  string
}
```

```
type SongRef struct {
    *Artist       // embedded pointer --- SongRef borrows Artist data
    Title  string
}
```

Use an **embedded value** when the outer struct fully owns the embedded data and you want simple value-copy semantics.

Use an **embedded pointer** when multiple outer structs share the same embedded object, or when the embedded type is large and you want to avoid copying it.

Trap: A zero-value `SongRef` has a `nil` `*Artist` pointer. Accessing any promoted field or method on a nil embedded pointer causes a runtime panic. Always initialize the embedded pointer before use:

```
sr := SongRef{
    Artist: &Artist{Name: "Angoscia", Country: "Italy"},
    Title:  "Emerald Triangle 2012",
}
```

Name Collisions

If two embedded types define a field or method with the same name, the compiler reports an ambiguity error the moment you try to access the name without qualification.

```
type Meta struct {
    Title string
}

type Track struct {
    Meta          // has Title
    Title  string // also has Title
}

t := Track{Meta: Meta{Title: "metadata title"}, Title: "track title"}
fmt.Println(t.Title)       // "track title" --- outer field wins (no ambiguity here)
fmt.Println(t.Meta.Title)  // "metadata title" --- must qualify to reach embedded field
```

When the outer type itself declares `Title`, that field shadows the promoted one — no ambiguity, outer wins. When two embedded types both promote the same name and the outer type does not declare it, the compiler errors on any unqualified access:

```
type A struct{ Val int }
type B struct{ Val int }

type C struct {
    A
    B
}

c := C{}
fmt.Println(c.Val)   // compile error: ambiguous selector c.Val
fmt.Println(c.A.Val) // OK
fmt.Println(c.B.Val) // OK
```

Embedding vs Inheritance

Java's class inheritance is an *is-a* relationship: a `FeaturedTrack` that extends `Track` is substitutable wherever a `Track` is expected. Go embedding is a *has-a* relationship: a `FeaturedTrack` that embeds `Track` gets `Track`'s promoted fields and methods, but is not substitutable for a `Track`.

Side-by-Side Comparison

```java
// Java: inheritance
class FeaturedTrack extends Track {
    String feature;

    FeaturedTrack(String title, String artist, int bpm, String feature) {
        super(title, artist, bpm);
        this.feature = feature;
    }

    @Override
    public String toString() {
        return super.toString() + " ft. " + feature;
    }
}

// A FeaturedTrack IS-A Track --- substitutable.
Track t = new FeaturedTrack("Gamemaster", "Matt Darey & Lost Tribe", 97, "Alizée");
```

```go
// Go: embedding
type FeaturedTrack struct {
    Track              // has-a Track, not is-a Track
    Feature string
}

func (ft *FeaturedTrack) String() string {
    return ft.Track.String() + " ft. " + ft.Feature
}
```

`FeaturedTrack.String` uses a pointer receiver because the embedded `Track.String` does, and the all-pointer-receiver idiom applies to the outer type too.

```go
ft := FeaturedTrack{
    Track:   Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 97},
    Feature: "Alizée",
}

fmt.Println(ft.Title)    // Gamemaster --- promoted
fmt.Println(ft.String()) // Gamemaster by Matt Darey & Lost Tribe (97 BPM) ft. Alizée
```

But this does **not** compile:

```go
var t Track = ft  // compile error: cannot use FeaturedTrack as Track
```

A `FeaturedTrack` is not a `Track`. It has a `Track` inside, but Go's type system does not consider that inheritance.

What You Get and What You Don't

What embedding gives you:

- Promoted fields: `ft.Title`, `ft.Artist`, `ft.BPM` work without qualification.

- Promoted methods: `ft.ScaleBPM(1.05)` delegates to the embedded `Track`'s method.
- Less boilerplate: no need to write forwarding methods by hand.

What embedding does not give you:

- Substitutability: a `FeaturedTrack` cannot be passed where a `Track` is expected.
- Virtual dispatch: there is no override mechanism; the outer type's method simply shadows the inner one.

How Interfaces Fill the Gap

Go fills the polymorphism gap with interfaces (covered in Chapter 8). If both `Track` and `FeaturedTrack` implement the same interface, they can be used interchangeably through that interface — regardless of their struct relationship. Method promotion has a powerful consequence here: because an embedded type's methods are promoted to the outer type, the outer type automatically satisfies any interface the embedded type satisfies (see Chapter 8) — you can embed a type purely to inherit its interface implementation for free.

```
type Playable interface {
    String() string
}

// Both *Track and *FeaturedTrack provide String() ---
// both satisfy Playable, independently.
func announce(p Playable) {
    fmt.Println("Now playing:", p.String())
}

announce(&t)   // *Track satisfies Playable
announce(&ft)  // *FeaturedTrack also satisfies Playable
```

Tip: The Go proverb is "favor composition over inheritance." Go takes that further: composition via embedding is the *only* option. Once you internalize it, you will find the explicit has-a relationship easier to reason about than deep inheritance hierarchies.

A Realistic Embedding Example: LoggedPlaylist

Embedding shines when you want to extend the behavior of an existing type without modifying it. A common pattern is to wrap a type with one that adds cross-cutting concerns — logging, metrics, caching — by embedding the original and selectively overriding methods.

```
import (
    "fmt"
    "log"
)

type LoggedPlaylist struct {
    Playlist                    // embeds all of Playlist's fields and methods
}

func NewLoggedPlaylist(name string) *LoggedPlaylist {
    return &LoggedPlaylist{Playlist: Playlist{Name: name}}
}

// Add wraps Playlist.Add with a log line.
func (lp *LoggedPlaylist) Add(t Track) {
```

```
    log.Printf("adding %q to %q", t.Title, lp.Name)
    lp.Playlist.Add(t) // delegate to the embedded method
}

lp := NewLoggedPlaylist("Late Night Vibes")
lp.Add(Track{Title: "Flaming June", Artist: "BT", BPM: 118})
// 2026/05/28 00:00:00 adding "Flaming June" to "Late Night Vibes"

fmt.Println(lp.Len())          // 1  --- promoted from Playlist
fmt.Println(lp.AverageBPM())   // 118 --- promoted from Playlist
```

`LoggedPlaylist` inherits `Len` and `AverageBPM` for free via promotion. It only needs to define `Add` — the one method it wants to wrap.

Trap: When `LoggedPlaylist.Add` calls `lp.Playlist.Add(t)`, it must use the explicit qualified path `lp.Playlist.Add`. Writing `lp.Add(t)` inside the method would call `LoggedPlaylist.Add` recursively and loop forever.

Try It

Type this in and run it: it pulls together every idea in this chapter — pointer receivers, a `New*` factory, embedding via promotion, a wrapped method, and `defer` for end-of-function cleanup. The `LoggedPlaylist` embeds a `*Playlist`, gets `Len` for free, and overrides only `Add`.

```
package main

import (
    "fmt"
    "log"
)

type Track struct {
    Title  string
    Artist string
    BPM    int
}

func (t *Track) String() string { // pointer receiver, consistent across the type
    return fmt.Sprintf("%s by %s (%d BPM)", t.Title, t.Artist, t.BPM)
}

type Playlist struct {
    Name   string
    tracks []*Track
}

func NewPlaylist(name string) *Playlist { // New* factory in place of a constructor
    return &Playlist{Name: name}
}

func (p *Playlist) Add(t *Track) { // pointer receiver: mutates the slice
    p.tracks = append(p.tracks, t)
}
```

```
func (p *Playlist) Len() int { // pointer receiver for consistency
    return len(p.tracks)
}

type LoggedPlaylist struct {
    *Playlist // embedded pointer: promotes Len and friends
}

func (lp *LoggedPlaylist) Add(t *Track) { // wraps the embedded Add
    log.Printf("queueing %q", t.Title)
    lp.Playlist.Add(t) // explicit path avoids infinite recursion
}

func main() {
    defer fmt.Println("done") // runs last, like a cleanup step

    lp := &LoggedPlaylist{Playlist: NewPlaylist("Fiesta")}
    lp.Add(&Track{Title: "Emerald Triangle 2012", Artist: "Angoscia", BPM: 127})
    lp.Add(&Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 97})

    fmt.Println("tracks:", lp.Len()) // promoted from Playlist
    for _, t := range lp.tracks {
        fmt.Println(t) // calls (*Track).String automatically
    }
}
```

The `log` lines carry a timestamp, but the rest is deterministic: a `tracks: 2` count, the two tracks formatted through `String`, and `done` printed last by the deferred call.

Try these modifications:

- Add an `AverageBPM` method on `*Playlist` and call it through `lp` — notice it is promoted for free.
- Change `LoggedPlaylist.Add` to mistakenly call `lp.Add(t)` instead of `lp.Playlist.Add(t)` and watch the stack overflow.
- Swap the embedded `*Playlist` for a value `Playlist` and see which calls still compile.

Key Points

- A method is a function with a receiver declared between `func` and the method name.
- A value receiver `(t T)` gets a copy; a pointer receiver `(t *T)` gets a pointer to the original and can mutate it.
- If any method on a type uses a pointer receiver, use pointer receivers for all methods on that type.
- The method set of `*T` includes all methods of `T`; the method set of `T` includes only value-receiver methods.
- Go auto-takes the address for pointer-receiver calls on addressable values and auto-dereferences for value-receiver calls on pointers.
- Go has no constructor syntax; the idiomatic replacement is a `New*` factory function.
- Factory functions return `*T` so the caller can immediately use pointer-receiver methods.
- Factory functions can return `(T, error)` or `(*T, error)` to signal construction failures.
- `go doc` and IDE tooling recognize the `New*` naming convention.
- Go has no destructor syntax; the GC reclaims heap memory automatically.
- Use `defer resource.Close()` immediately after a successful resource acquisition for deterministic cleanup.
- `defer` is Go's equivalent of Java's try-with-resources; multiple defers form a LIFO cleanup stack.
- `runtime.SetFinalizer` exists but is rarely correct; prefer `defer`.

- Embedding includes one struct type inside another by naming only the type, with no field name.
- Embedded fields and methods are promoted to the outer type; you can access them without qualification.
- An embedded value gives full ownership; an embedded pointer shares the embedded data.
- A zero-value struct with an embedded pointer has a nil embedded pointer; accessing promoted members panics.
- When two embedded types have the same field or method name and the outer type does not shadow it, any unqualified access is a compile error; resolve it with an explicit path.
- Go embedding is a has-a relationship, not an is-a relationship; a `FeaturedTrack` cannot be used where a `Track` is expected.
- Interfaces fill the polymorphism gap: any type that provides the required methods satisfies the interface, regardless of its embedding structure.
- "Favor composition over inheritance" is the Go proverb; embedding is the language's only code-reuse mechanism.

Exercises

1. **Think about it:** In Java, a class bundles data and behavior together and inheritance lets you share both across a type hierarchy. Go separates data (struct), behavior (methods), and code reuse (embedding) into three distinct mechanisms, and interfaces handle polymorphism independently of all three. What advantages does Go's separated approach offer over Java's unified class model? Can you think of a scenario where Java's approach is simpler or more convenient?

2. **What does this print?**

```go
package main

import "fmt"

type Base struct {
    ID int
}

func (b Base) Describe() string {
    return fmt.Sprintf("Base ID=%d", b.ID)
}

type Widget struct {
    Base
    Color string
}

func main() {
    w := Widget{
        Base:  Base{ID: 42},
        Color: "blue",
    }
    fmt.Println(w.ID)
    fmt.Println(w.Color)
    fmt.Println(w.Describe())
    fmt.Println(w.Base.Describe())
}
```

3. **Calculation:** Given the types below, count the methods in each method set.

```go
type Track struct {
    Title  string
    Artist string
}

func (t *Track) String() string     { /* ... */ }
func (t *Track) ScaleBPM(f float64) { /* ... */ }
func (t Track) IsLong() bool        { /* ... */ }

type FeaturedTrack struct {
    Track
    Feature string
}

func (ft *FeaturedTrack) String() string { /* ... */ }
```

a. How many methods are in the method set of `Track` (the value type)?
b. How many methods are in the method set of `*Track`?
c. How many methods are in the method set of `FeaturedTrack` (the value type), counting promoted methods?
d. How many methods are in the method set of `*FeaturedTrack`, counting promoted methods?

4. **Where is the bug?** The following program panics at runtime. Identify the exact line that panics, explain why, and describe how to fix it.

```go
package main

import "fmt"

type Artist struct {
    Name string
}

func (a Artist) Label() string {
    return "Artist: " + a.Name
}

type Song struct {
    *Artist
    Title string
}

func main() {
    s := Song{Title: "Out Of The Blue"}
    fmt.Println(s.Title)
    fmt.Println(s.Label()) // line A
}
```

5. **Write a program:** Define a struct `Counter` with a single `int` field `Value`. Write a `New*` constructor that accepts a starting value and returns a `*Counter`. Add three pointer-receiver methods: `Increment()` that adds 1, `Reset()` that sets `Value` to zero, and `String() string` that returns the current value formatted as `"count: N"`. In `main`, create a `Counter` with `NewCounter(10)`, increment it three times, print it, reset it, and print it again. Use `defer` to print `"done"` at the end of `main`, so that "done" appears as the very last line of output.

6. **Where is the bug?** The following program does not compile. Explain the exact reason the compiler

rejects it, and describe the smallest change that makes it work.

```
package main

import "fmt"

func (n int) IsFast() bool {
    return n >= 125
}

func main() {
    bpm := 128
    fmt.Println(bpm.IsFast())
}
```

Google. 2024. “Go Best Practices: Package Size.” Google’s Go Style Guide. https://google.github.io/styleguide/go/best-practices.html#package-size.