Go Intermediate

A Complete 3-Chapter Course

Topics covered:

Structs in Full, Methods, and Pointers

Interfaces

Goroutines and Channels

Exercises: 9 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · Quick-reference tables

Table of Contents

CHAPTER 1 OF 3

Structs in Full, Methods, and Pointers

COURSE 2 · CH 1

Structs in Full, Methods, and Pointers

Go's real answer to "object" — a fixed-shape type, functions attached to it, and the one new concept that makes both work together

Fundamentals Chapter 7 previewed `struct` in passing. This chapter covers it properly: defining one, attaching behaviour to it with **methods**, and the **pointer** concept that decides whether a method can actually change the struct it's called on. Go has no `class` keyword and no `this` — methods and pointers together are how Go achieves what JavaScript's object methods did with `this` (Fundamentals Chapter 7).

Defining and Creating a Struct

```
type Person struct {
        Name string
        Age  int
}

p := Person{Name: "Philip", Age: 35}

fmt.Println(p.Name)   // "Philip"
p.Age = 36             // fields can be reassigned after creation
```

`type Person struct { ... }` defines a new named type with a fixed set of fields. Field access uses the same dot notation as JavaScript object properties — but unlike a map (Chapter 7), a struct's fields are fixed at compile time: there's no way to add an unplanned field later, and each field can have its own distinct type.

CAPITALISED FIELDS ARE INTENTIONAL

`Name` and `Age` start with capital letters deliberately — in Go, capitalisation controls visibility outside the current package (covered fully later). Lowercase struct fields still work everywhere in this course's single-file examples, but capitalised fields are the convention for anything meant to be used elsewhere.

Methods — Functions Attached to a Type

```
func (p Person) Greet() string {
        return "Hi, I'm " + p.Name
}

p := Person{Name: "Philip", Age: 35}
fmt.Println(p.Greet())   // "Hi, I'm Philip"
```

`(p Person)` between `func` and the method name is the **receiver** — it's what makes this an ordinary function into a method that can be called as `p.Greet()`. `p` inside the method body plays the same role JavaScript's `this` played inside an object method (Fundamentals Chapter 7) — but it's an explicit, named parameter here, not an implicit keyword.

Pointers — Why Greet() Can't Change p, But Something Else Can

```
age := 35
agePointer := &age          // & gets the memory address of age

fmt.Println(agePointer)     // 0xc0000140a0 — an address, not a value
fmt.Println(*agePointer)    // 35 — * "dereferences": follows the pointer to the actual value

*agePointer = 36
fmt.Println(age)          // 36 — changing through the pointer changed the original
```

`&age` ("address of") produces a **pointer** — a value that points to where `age` actually lives in memory, rather than a copy of `35` itself. `*agePointer` ("dereference") goes the other way: follow the pointer back to the real value. This is a genuinely new concept with no real JavaScript equivalent — every JavaScript variable is already accessed "directly," with no separate address-vs-value distinction to think about.

Why This Matters for Methods

```go
func (p Person) HaveBirthday() {
        p.Age++   // only changes the COPY inside this method
}

func (p *Person) HaveBirthdayPointer() {
        p.Age++   // changes the REAL struct, through the pointer
}

person := Person{Name: "Philip", Age: 35}

person.HaveBirthday()
fmt.Println(person.Age)          // 35 — unchanged!

person.HaveBirthdayPointer()
fmt.Println(person.Age)          // 36 — actually changed
```

A receiver of type `Person` (no `*`) receives a brand-new **copy** of the struct — any changes inside the method vanish once it returns, the value-passing behaviour every Go function has by default. A receiver of type `*Person` (a **pointer receiver**) receives a pointer to the original struct instead, so changes made through it persist. Go automatically handles the `&` when calling `person.HaveBirthdayPointer()` — no manual `&person.HaveBirthdayPointer()` needed.

IF A METHOD NEEDS TO CHANGE THE STRUCT, IT NEEDS A POINTER RECEIVER

This is the single most common beginner mistake with Go structs: writing a method that's supposed to update a field, using a plain (non-pointer) receiver, and then being confused why the change doesn't stick outside the method. If a method modifies `p.Field` and that change should be visible afterward, the receiver must be `*Person`, not `Person`.

JAVASCRIPT	GO
`{ name: "Philip", age: 35 }`	`type Person struct { Name string; Age int }`
`greet: function() { ...this... }`	`func (p Person) Greet() string { ...p... }`
`this (implicit)`	`Receiver variable (explicit, named by you)`
`Objects are always reference-shared`	`Plain receiver = copy; pointer receiver (*Type) = shared`
`No address/value distinction`	`&value (address), *pointer (dereference)`

Coding Challenges

CHALLENGE 1

Define a struct Book with fields Title (string), Pages (int). Create one, then write a method (b Book) Describe() string that returns a sentence combining both fields. Print the result of calling Describe().

View solution

CHALLENGE 2

Define a struct Account with field Balance (float64). Write a method (a *Account) Deposit(amount float64) using a pointer receiver that adds amount to Balance. Create an account, call Deposit twice with different amounts, and print Balance after each call to confirm it actually changes.

View solution

CHALLENGE 3

Write a function double(n *int) that takes a pointer to an int and doubles the value it points to (using dereferencing, not a return value). Declare a variable, pass its address to double, and print the variable afterward to confirm it changed.

View solution

CHAPTER 1 QUICK REFERENCE (INTERMEDIATE)

- **type Name struct { Field type }** — defines a fixed-shape record type
- **func (receiver Type) MethodName() { }** — attaches a method to a type
- **&value** — gets a pointer (the address) to a value
- ***pointer** — dereferences a pointer, accessing/changing the real value
- **Plain receiver (p Type)** — method gets a copy; changes don't persist
- **Pointer receiver (p *Type)** — method gets the original; changes DO persist
- **Go calls automatically handle &** when invoking a pointer-receiver method on a plain variable
- **Next chapter:** interfaces — how Go achieves polymorphism without classes or inheritance

CHAPTER 2 OF 3

Interfaces

COURSE 2 · CH 2

Interfaces: Polymorphism Without Classes or Inheritance

Go has no "extends" — types qualify for a shared role just by having the right methods, with no declaration required

Chapter 1 gave `Person` a method. This chapter asks: how does code write a function that works on *any* type that has a particular method, without caring exactly what that type is? Go's answer is the **interface** — and Go's interfaces work in a way that has no real JavaScript equivalent, since JavaScript has no static types to satisfy in the first place.

Defining an Interface

```
type Shape interface {
        Area() float64
}
```

An interface lists method **signatures** only — no implementation, no fields. `Shape` says: "anything with an `Area() float64` method counts as a `Shape`." Nothing else about the type matters.

Implementing an Interface — Implicitly

```
type Circle struct {
        Radius float64
}

func (c Circle) Area() float64 {
        return 3.14159 * c.Radius * c.Radius
}

type Rectangle struct {
        Width, Height float64
}

func (r Rectangle) Area() float64 {
        return r.Width * r.Height
}
```

Neither `Circle` nor `Rectangle` mentions `Shape` anywhere. There's no `implements Shape` keyword in Go at all — a type satisfies an interface automatically, just by having every method the interface requires, with matching signatures. This is called **structural typing**: it's about shape, not declared relationship.

> **NO "IMPLEMENTS" KEYWORD — THIS IS DELIBERATE**
>
> A type can satisfy a brand-new interface written months after the type itself was defined, with zero changes to the original type. This is the core reason Go interfaces are described as "implicit" — the connection only exists at the point something actually requires it.

Writing a Function That Accepts the Interface

```
func describe(s Shape) {
        fmt.Printf("Area: %.2f\n", s.Area())
}

c := Circle{Radius: 5}
r := Rectangle{Width: 4, Height: 6}

describe(c)   // Area: 78.54
describe(r)   // Area: 24.00
```

`describe` takes a `Shape` — it works on `Circle`, `Rectangle`, or any future type with an `Area()` method, all through one function. This is **polymorphism**: one piece of code, many concrete types, achieved here with no class hierarchy, no `extends`, and no inheritance chain at all.

A Slice of an Interface Type

```
shapes := []Shape{c, r}

for _, shape := range shapes {
        fmt.Printf("%.2f\n", shape.Area())
}
// 78.54
// 24.00
```

`[]Shape` can hold a mix of any concrete types that satisfy `Shape` — `Circle` and `Rectangle` side by side in the same slice, the same loop, the same `Area()` call. This is the most common real-world use for interfaces: a heterogeneous collection processed uniformly.

The Empty Interface and any

```
func printAnything(value any) {
        fmt.Println(value)
}

printAnything(42)
printAnything("hello")
printAnything(Circle{Radius: 2})
```

`any` (an alias for the older `interface{}`) is an interface with **zero** required methods — literally everything satisfies it, since there's nothing to satisfy. This is the closest Go gets to JavaScript's "accepts anything" flexibility, but it's used sparingly: reaching for `any` too often throws away the type-checking that makes Go, Go.

> **ANY ISN'T A FREE PASS BACK TO JAVASCRIPT-STYLE TYPING**
>
> A function receiving `any` doesn't know what's actually inside without a separate runtime check (a "type assertion" or "type switch," beyond this chapter's scope). Reaching for `any` as a default habit defeats the purpose of Go's type system — it should be the exception, not the rule.

CONCEPT	JAVASCRIPT	GO
Shared behaviour across types	Duck typing — works at runtime if methods exist	Interfaces — checked at COMPILE time
Declaring a relationship	class X extends Y / implements Z	Nothing — satisfied implicitly
"Accepts anything"	Default behaviour (no static types)	any — explicit opt-out, used sparingly

Coding Challenges

CHALLENGE 1

Define an interface Speaker with a method Speak() string. Define two structs, Dog and Cat, each with a Speak() method returning an appropriate sound. Write a function announce(s Speaker) that prints the result of Speak(), and call it with both a Dog and a Cat.

View solution

CHALLENGE 2

Using the Shape interface, Circle, and Rectangle from this chapter, create a slice []Shape containing at least 3 shapes (mixing circles and rectangles), then loop over it printing each one's area. Add up all the areas into a single total, printed at the end.

View solution

CHALLENGE 3

Write a function printAll(items []any) that takes a slice of any and prints each item on its own line. Call it with a slice containing a mix of an int, a string, and a Circle value.

View solution

CHAPTER 2 QUICK REFERENCE

- **type Name interface { Method() returnType }** — lists required method signatures only
- **No "implements" keyword** — a type satisfies an interface just by having matching methods
- **Structural typing** — the relationship is about shape, never declared explicitly
- **func f(x InterfaceType)** — accepts ANY concrete type that satisfies the interface
- **[]InterfaceType** — a slice that can mix different concrete types together
- **any (interface{})** — zero required methods; satisfied by literally everything
- **Polymorphism in Go** — achieved via interfaces, with no classes or inheritance chains
- **Next chapter:** goroutines and channels — Go's built-in approach to concurrency

CHAPTER 3 OF 3

Goroutines and Channels

COURSE 2 · CH 3

Goroutines and Channels

Go's built-in answer to "do several things at once" — lightweight, and designed to talk to each other safely

Fundamentals Chapter 10 covered JavaScript's `async` / `await` — useful for waiting on one thing at a time, like a single network request, without ever running two pieces of JavaScript truly simultaneously. Go's concurrency is fundamentally different: a Go program can run genuinely many things **at once**, using **goroutines**, and **channels** to let them communicate safely.

Starting a Goroutine with go

```
func sayHello() {
        fmt.Println("Hello from a goroutine!")
}

func main() {
        go sayHello()       // runs concurrently, doesn't block main()
        fmt.Println("Hello from main!")
        time.Sleep(100 * time.Millisecond)   // give the goroutine time to actually run
}
```

The keyword `go` placed before any function call launches it as a **goroutine** — a lightweight, independently-running unit of execution — and immediately continues with the next line, without waiting for it to finish. This is similar in spirit to JavaScript's "doesn't block" behaviour from Chapter 10's `setTimeout`, but a goroutine genuinely can run in parallel with everything else, on multi-core hardware.

MAIN() DOESN'T WAIT FOR GOROUTINES AUTOMATICALLY

If `main()` returns before a goroutine finishes, the goroutine is simply abandoned — there is no equivalent of JavaScript's event loop keeping things alive until they're done. The `time.Sleep` above is a crude workaround used only for this introductory example; real code uses the synchronization tools below instead.

Channels — Sending Values Between Goroutines

```go
func calculate(n int, results chan int) {
        results <- n * n   // send n² into the channel
}

func main() {
        results := make(chan int)

        go calculate(5, results)

        value := <-results   // receive — BLOCKS until something arrives
        fmt.Println(value)      // 25
}
```

`make(chan int)` creates a **channel** — a typed pipe that goroutines use to send and receive values safely. `results <- n*n` sends a value in; `<-results` receives one out. Receiving from a channel **blocks** — the receiving code waits patiently until a value actually arrives, which is exactly what removes the need for `time.Sleep` guesswork from the first example.

Running Several Goroutines and Collecting Results

```go
func square(n int, results chan int) {
        results <- n * n
}

func main() {
        numbers := []int{2, 3, 4}
        results := make(chan int, len(numbers))   // buffered channel

        for _, n := range numbers {
                go square(n, results)
        }

        for i := 0; i < len(numbers); i++ {
                fmt.Println(<-results)
        }
}
// 4, 9, 16 — order is NOT guaranteed
```

`make(chan int, len(numbers))` creates a **buffered** channel that can hold several values without anything having to receive immediately — letting all three goroutines send without waiting on each other. The receiving loop then collects exactly as many results as goroutines were launched. The order they arrive in is not guaranteed — whichever `square` finishes first sends first.

sync.WaitGroup — Waiting for a Group of Goroutines

```go
import "sync"

func worker(id int, wg *sync.WaitGroup) {
        defer wg.Done()   // runs when worker() returns, however it returns
        fmt.Println("Worker", id, "done")
}

func main() {
        var wg sync.WaitGroup

        for i := 1; i <= 3; i++ {
                wg.Add(1)
                go worker(i, &wg)
        }

        wg.Wait()   // blocks until all 3 have called Done()
        fmt.Println("All workers finished")
}
```

`WaitGroup` is the real-world tool for "wait until everything launched is finished" — `Add(1)` registers one more goroutine to wait for, each goroutine calls `Done()` when finished, and `Wait()` blocks until the count returns to zero. `defer wg.Done()` guarantees `Done()` runs no matter how the function exits, even if it returns early or panics.

DEFER — RUN THIS JUST BEFORE THE FUNCTION RETURNS

`defer` schedules a statement to run right before its surrounding function exits, regardless of which `return` path is taken. It's used constantly alongside resources that need cleanup (closing a file, unlocking something) — `wg.Done()` here is the most common beginner-facing example.

JAVASCRIPT	GO
Single-threaded event loop, never truly parallel	Goroutines genuinely run in parallel on multi-core hardware
await / .then() to wait for a result	<-channel blocks until a value arrives
Promise.all([...]) to wait for several	sync.WaitGroup — Add/Done/Wait
No raw threads exposed to the developer	go keyword launches a real concurrent goroutine directly

Coding Challenges

CHALLENGE 1

Write a function cube(n int, results chan int) that sends n^3 into results. In main, create an unbuffered channel, launch cube(4, results) as a goroutine, receive the value, and print it.

View solution

CHALLENGE 2

Given numbers := []int{1, 2, 3, 4, 5}, launch a goroutine per number that sends its square into a buffered channel sized to len(numbers). Receive all 5 results in a loop, summing them into a total printed at the end.

View solution

CHALLENGE 3

Write a function worker(id int, wg *sync.WaitGroup) that prints "Worker started" and "Worker finished" with the id, using defer wg.Done(). Launch 4 workers using a sync.WaitGroup, then print "All done" only after wg.Wait() returns.

View solution

CHAPTER 3 QUICK REFERENCE

- **go functionCall()** — launches a goroutine; continues immediately without waiting
- **make(chan Type)** — unbuffered channel; sending/receiving blocks until matched
- **make(chan Type, n)** — buffered channel; can hold up to n values without blocking
- **channel <- value** — send; **<-channel** — receive (blocks until a value arrives)
- **sync.WaitGroup** — Add(n) to register, Done() when finished, Wait() to block until all are done
- **defer statement** — runs just before the surrounding function returns, however it returns
- **main() does NOT wait for goroutines automatically** — use channels or WaitGroup, not Sleep, in real code
- **This completes this Go course's planned chapters.** Advanced topics (generics, context, testing) would follow in a Course 3.