Go Fundamentals

A Complete 7-Chapter Course

Topics covered:

Basics — package main, go run/build, fmt

Variables, Basic Types, and := vs var

Operators, Arithmetic Type Rules, and Control Flow

Loops: for as Go's Only Loop Keyword

Functions: Multiple Returns, Named Returns, Variadic Params

Arrays and Slices

Maps

Exercises: 21 hands-on challenges with sample solutions

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

Table of Contents

CHAPTER 1 OF 7

Basics — package main, go run/build, fmt

COURSE 1 · CH 1

What Go Is, Compiling and Running, and Your First Program

A compiled, statically-typed language built at Google for simplicity and speed — starting with the smallest program that runs

Go (often called **Golang** to make it searchable) is a **compiled** language — source code is translated into a standalone executable before it runs, rather than being interpreted line-by-line in a browser console. This is the single biggest difference to keep in mind throughout this course: there's no browser, no `console.log`-and-refresh workflow — Go code is built, then run, as a real program.

Installing Go and Checking It Works

```
# In a terminal, after installing Go:
go version
# go version go1.22.0 windows/amd64
```

If that prints a version number, Go is installed and ready to use from the command line.

The Smallest Go Program

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, Go!")
}
```

Every runnable Go file has exactly these three pieces. `package main` marks this file as a program (not a reusable library); `import "fmt"` pulls in the standard formatting/printing package; `func main()` is the entry point — execution always starts here, the direct equivalent of the top-level code that ran automatically in every JavaScript example.

FMT.PRINTLN IS GO'S CONSOLE.LOG

`fmt.Println(...)` prints a line of output to the terminal. It's the most-used function in this entire course, exactly as `console.log` was throughout the JavaScript course.

Running a Go File

```
# Save the code above as hello.go, then:
go run hello.go
# Hello, Go!
```

`go run` compiles the file into a temporary executable and runs it immediately in one step — the fastest way to try something out. There's also a separate, two-step workflow for producing a permanent program:

```
go build hello.go    # produces hello.exe (Windows) or hello (Mac/Linux)
./hello               # runs the compiled program directly, no Go installation needed to run it
```

`go build` is what actually happens when shipping a real program — the resulting file runs on its own, with the Go toolchain itself no longer required on the machine that runs it.

GO IS STRICT ABOUT UNUSED THINGS

Go refuses to compile if a variable is declared but never used, or a package is imported but never referenced. This feels harsh coming from JavaScript, where unused variables are silently ignored — but it's deliberate, catching real mistakes (like a typo'd variable name) immediately instead of letting them hide.

Comments

```
// A single-line comment

/*
   A multi-line comment,
   identical syntax to JavaScript/CSS
*/
```

Comment syntax is identical to JavaScript's — `//` for a single line, `/* */` for a block.

Printing Multiple Things

```
fmt.Println("The answer is", 42)        // The answer is 42
fmt.Printf("The answer is %d\n", 42)    // The answer is 42 — formatted, like a template literal
```

`Println` auto-spaces and joins multiple arguments, then adds a newline. `Printf` uses placeholder verbs (`%d` for a number, `%s` for a string, `%v` for "any value") inside a format string — closer in spirit to a JavaScript template literal, but with explicit type placeholders instead of `${ }`.

CONCEPT	JAVASCRIPT	GO
Run code	Browser / Node, instantly	go run file.go (compiles first)
Print output	console.log(...)	fmt.Println(...)
Entry point	Top-level script code	func main() { }
Unused variables	Allowed, silently ignored	Compile error

Coding Challenges

CHALLENGE 1

Write a complete Go program (package main, import "fmt", func main) that prints your name and a short greeting on two separate lines using two calls to fmt.Println.

View solution

CHALLENGE 2

Write a program that uses fmt.Printf with %d to print "Year: 2026" and %s to print "Language: Go" — two separate Printf calls, each using a placeholder rather than concatenation.

View solution

CHALLENGE 3

Write a program that declares a variable message with the value "Compiled and ready", but deliberately never uses it (no Println call referencing it). Try go run on it, note the exact compiler error, then fix it by actually printing the variable.

View solution

CHAPTER 1 QUICK REFERENCE

- **package main** — marks a file as a runnable program
- **import "fmt"** — brings in the formatting/printing package
- **func main() { }** — the entry point; execution always starts here
- **go run file.go** — compile and run immediately, for quick testing
- **go build file.go** — produce a standalone executable to ship
- **fmt.Println(...)** — print a line, auto-spaced and newline-terminated
- **fmt.Printf("...%d...\n", x)** — formatted printing with explicit placeholders
- **Unused variables/imports are compile errors**, not warnings — Go enforces this strictly
- **Next chapter:** variables, Go's basic types, and := vs var

CHAPTER 2 OF 7

Variables, Basic Types, and := vs var

COURSE 1 · CH 2

Variables, Basic Types, and := vs var

Go is statically typed — every variable has a fixed type, decided either explicitly or by what's assigned to it

Chapter 1's `message := "Compiled and ready"` already used a variable without explaining it. JavaScript's `let` / `const` (any value, any type, can hold anything later) has no direct equivalent here — Go variables are **statically typed**: once a variable is created as a string, it can never later hold a number.

Declaring with var and an Explicit Type

```
var name string = "Philip"
var age int = 35
var isStudent bool = true
```

`var name string = "Philip"` reads as: declare a variable called `name`, of type `string`, set to `"Philip"`. The type is written explicitly between the name and the value — the opposite order to `let`, where no type appears at all.

The Short Declaration Operator :=

```
name := "Philip"   // type "string" is inferred automatically
age := 35          // type "int" is inferred
isStudent := true  // type "bool" is inferred
```

`:=` declares a new variable AND infers its type from the value on the right, in one step — far closer to JavaScript's `const x = ...` in everyday feel. It's the form used almost everywhere inside functions; `var` with an explicit type is mostly reserved for cases where no value is assigned yet, or the type genuinely needs to be different from what would be inferred.

:= ONLY WORKS FOR NEW VARIABLES

> `name := "Philip"` followed later by another `name := "Sam"` in the same scope is a compile error — `:=` declares, it doesn't just assign. To change an existing variable's value, drop the colon: `name = "Sam"`.

Go's Core Basic Types

```
count := 42            // int — whole numbers
price := 19.99         // float64 — decimal numbers
label := "Hello"       // string — text, always double-quoted
active := true         // bool — true or false, lowercase, no quotes
```

These four cover almost all everyday Go code: `int`, `float64`, `string`, `bool`. Unlike JavaScript's single general-purpose `number` type, Go separates whole numbers (`int`) from decimals (`float64`) — mixing them directly causes a compile error, covered next chapter.

Zero Values — Go Never Leaves a Variable Truly Empty

```
var count int       // 0 — the "zero value" for int
var label string    // "" — empty string, the zero value for string
var active bool     // false — the zero value for bool
```

Declaring a variable with `var` and no value doesn't leave it as `undefined` the way JavaScript would — Go automatically gives it a sensible default for its type, called the **zero value**. There is no Go equivalent of `undefined`; every variable always holds a real, usable value of its type from the moment it's declared.

Declaring Several Variables Together

```
name, age := "Philip", 35

var (
        width  float64 = 12.5
        height float64 = 4.0
)
```

Multiple variables can be declared on one line with `:=`, matching values to names left to right. The `var ( ... )` block groups several explicit declarations together — useful for constants and configuration-style values declared at the top of a file, outside any function.

JAVASCRIPT	GO	NOTES
`let x = 5;`	`x := 5`	Type inferred automatically in both
`let x; // undefined`	`var x int // 0`	Go has no "undefined" — zero values fill the gap
`x can later hold a string`	`x cannot — fixed type forever`	Static typing is the core difference
`const for "shouldn't reassign"`	`No direct equivalent for variables`	Go's const is for fixed, compile-time-known values only

Coding Challenges

CHALLENGE 1

Write a program that declares title (string), pages (int), and inStock (bool) using := for each, then prints all three using a single fmt.Println call.

View solution

CHALLENGE 2

Write a program that declares var total int with no value, prints it (confirming it's 0, not an error), then assigns it 100 using = (not :=) and prints it again.

View solution

CHALLENGE 3

Write a program that declares city, population := "London", 8900000 on one line, then deliberately tries city := "Paris" again further down in main(). Run it, note the compiler error, then fix it using = instead of := for the second assignment.

View solution

CHAPTER 2 QUICK REFERENCE

- **var name type = value** — explicit declaration with an explicit type
- **name := value** — short declaration; type is inferred, only for NEW variables
- **name = value** (no colon) — reassigns an EXISTING variable
- **Core types:** int, float64, string, bool
- **Zero values:** int → 0, float64 → 0, string → "", bool → false (no "undefined" in Go)

- **Static typing** — a variable's type is fixed forever once declared
- **var (…)** block — group several explicit declarations together
- **Next chapter:** operators, arithmetic, and Go's strict rules about mixing numeric types

CHAPTER 3 OF 7

Operators, Arithmetic Type Rules, and Control Flow

COURSE 1 · CH 3

Operators, Arithmetic Type Rules, and Control Flow

Doing maths the strict way, comparing values, and branching with if/else and switch — no ternary operator here

Chapter 2 introduced `int` and `float64` as separate types. This chapter shows why that separation matters the moment arithmetic is involved, then covers the operators and branching structures used to make decisions — Go's version of Fundamentals JavaScript Chapter 3, with one notable omission: there is no ternary operator.

Arithmetic Operators

```
a := 10
b := 3

fmt.Println(a + b)   // 13
fmt.Println(a - b)   // 7
fmt.Println(a * b)   // 30
fmt.Println(a / b)   // 3 — integer division truncates, it does NOT round
fmt.Println(a % b)   // 1 — remainder
```

`+ - * / %` work as expected, but `/` between two `int` values performs **integer division** — the decimal part is discarded entirely, not rounded. `10 / 3` gives exactly `3`, never `3.33`, because both operands are whole numbers.

GO WILL NOT SILENTLY MIX INT AND FLOAT64

`var price float64 = 10 / 3` still produces `3` stored as a float — the division itself happened entirely in `int` first, because both `10` and `3` were untyped integer literals. To get a real decimal result, at least one operand must actually be a `float64`: `10.0 / 3` gives `3.3333...`. Mixing an actual `int` variable with a `float64` variable directly (`intVar / floatVar`) is a compile error — there is no automatic conversion, unlike JavaScript's number type.

Converting Between Types Explicitly

```
var count int = 7
var total float64 = 2

result := float64(count) / total   // 3.5 — count is converted to float64 first
```

`float64(count)` explicitly converts an `int` to a `float64` for that one expression. This explicitness — having to ask for a conversion rather than it happening automatically — is deliberate: it forces a decision about precision instead of letting a type mismatch hide silently.

Comparison and Logical Operators

```
age := 20

fmt.Println(age == 20)    // true
fmt.Println(age != 18)    // true
fmt.Println(age >= 18 && age < 65)   // true — && is "and"
fmt.Println(age < 13 || age > 100)   // false — || is "or"
```

`== != > < >= <=` and `&& ||` behave identically to JavaScript. One important difference: Go has no `===` — there's only one equality operator, `==`, since Go's static typing already guarantees both sides are the same type, removing the type-coercion ambiguity that `==` has in JavaScript.

if / else — No Parentheses, Braces Required

```
age := 16

if age >= 18 {
        fmt.Println("Adult")
} else if age >= 13 {
        fmt.Println("Teenager")
} else {
        fmt.Println("Child")
}
```

The condition itself has no surrounding parentheses — `if age >= 18`, not `if (age >= 18)` — but the curly braces are **mandatory**, even for a single statement. Omitting them is a compile error, unlike JavaScript where braces are optional for one-line bodies.

switch — Cleaner Than a Chain of else if

```go
day := 3

switch day {
case 1:
        fmt.Println("Monday")
case 2, 3, 4, 5:
        fmt.Println("Midweek")
default:
        fmt.Println("Weekend")
}
// Midweek
```

Go's `switch` needs no `break` at the end of each case — unlike JavaScript, it never "falls through" to the next case automatically. A single `case` can also list several values separated by commas (`case 2, 3, 4, 5:`), avoiding repeated cases entirely.

> **NO TERNARY OPERATOR IN GO**
>
> JavaScript's `condition ? a : b` has no Go equivalent at all — Go's designers left it out deliberately, favouring a full `if/else` even for simple cases, on the principle that it's more readable even though it's more verbose.

JAVASCRIPT	GO	NOTES
`if (x) { }`	`if x { }`	No parentheses around condition; braces always required
`x === y`	`x == y`	Only one equality operator — types are already guaranteed to match
`switch with break`	`switch, no break needed`	Go never falls through by default
`cond ? a : b`	`No equivalent`	Must use a full if/else
`5 / 2 === 2.5`	`5 / 2 == 2 (int)`	Integer division truncates; convert to float64 for a decimal result

Coding Challenges

CHALLENGE 1

Write a program with two int variables, total := 17 and count := 5. Print total / count (integer division), then print the same calculation converted properly to a float64 result using float64(), so it shows the real decimal value.

View solution

CHALLENGE 2

Write a program with score := 72. Use if/else if/else to print "Pass with distinction" (score >= 85), "Pass" (score >= 50), or "Fail" (anything else).

View solution

CHALLENGE 3

Write a program with grade := "B". Use a switch statement to print a description for "A" ("Excellent"), "B" or "C" together ("Satisfactory"), and a default case ("Needs improvement") for anything else.

View solution

CHAPTER 3 QUICK REFERENCE

- **+ - * / %** — standard arithmetic; int / int truncates instead of rounding
- **float64(x)** — explicit conversion; Go never converts numeric types automatically
- **== != > < >= <= && ||** — same meaning as JavaScript; only one equality operator (==)
- **if x { } else if y { } else { }** — no parentheses around the condition, braces always mandatory
- **switch x { case a: ... }** — no break needed, no automatic fall-through
- **case a, b, c:** — multiple values in one case, comma-separated
- **No ternary operator** — always use a full if/else instead
- **Next chapter:** loops — for is Go's only loop keyword, covering every JavaScript loop style

CHAPTER 4 OF 7

Loops: for as Go's Only Loop Keyword

COURSE 1 · CH 4

Loops: for as Go's Only Loop Keyword

One keyword, four shapes — covering everything JavaScript split across for, while, for...of, and for...in

JavaScript has four loop keywords: `for`, `while`, `for...of`, `for...in`. Go has exactly **one** — `for` — and reshapes it to cover every one of those cases. Recognising which "shape" of `for` is in use is the main new skill this chapter teaches.

Shape 1: The Classic Counting Loop

```
for i := 0; i < 5; i++ {
        fmt.Println(i)
}
// 0 1 2 3 4
```

This looks almost identical to JavaScript's `for (let i = 0; i < 5; i++)` — minus the parentheses and semicolons-as-separators staying the same. `i++` works exactly as before; Go also has `i--`, but notably no `++i` prefix form — increment/decrement are statements in Go, not expressions, so they're always written after the variable.

Shape 2: for as a while Loop

```
count := 0

for count < 3 {
        fmt.Println("Still counting:", count)
        count++
}
// Still counting: 0
// Still counting: 1
// Still counting: 2
```

Dropping the init and post clauses, leaving only a condition, turns `for` into the exact equivalent of JavaScript's `while` — there is no separate `while` keyword in Go at all, this is simply how it's written.

Shape 3: An Infinite Loop, with break

```
attempts := 0

for {
        attempts++
        fmt.Println("Attempt", attempts)
        if attempts == 3 {
                break
        }
}
```

`for` with absolutely nothing after it loops forever, until something inside explicitly `break`s out — the Go equivalent of JavaScript's `while (true) { ... break; }`. `continue` also works exactly as it does in JavaScript, skipping straight to the next iteration.

Shape 4: for...range — Iterating a Slice

```
fruits := []string{"apple", "banana", "cherry"}

for index, fruit := range fruits {
        fmt.Println(index, fruit)
}
// 0 apple
// 1 banana
// 2 cherry
```

`[]string{...}` is a **slice** — Go's closest equivalent to a JavaScript array, covered properly next chapter. `for index, value := range fruits` is the direct equivalent of JavaScript's `for...of`, except it always hands back both the index AND the value together — there's no separate index-only loop needed.

> **IGNORING THE INDEX WITH THE BLANK IDENTIFIER _**
>
> If only the value is needed, the index can be discarded with Go's "blank identifier": `for _, fruit := range fruits { ... }`. Go requires every declared variable to be used (Chapter 1), so `_` exists specifically to say "I know this value exists, I'm deliberately not using it."

Looping Over a Map (Go's Object Equivalent)

```go
scores := map[string]int{"maths": 88, "art": 95}

for subject, score := range scores {
        fmt.Println(subject, score)
}
```

The same `range` keyword also walks over a **map** (covered fully in Chapter 7) — Go's rough equivalent of a JavaScript object — handing back each key and value pair, the conceptual cousin of JavaScript's `for...in`.

> **MAP ITERATION ORDER IS NOT GUARANTEED**
>
> Unlike a slice, where `range` always visits elements 0, 1, 2... in order, ranging over a map visits entries in a deliberately randomised order every time the program runs. Code that depends on a specific order from a map will behave unpredictably — sort the keys first if order matters.

JAVASCRIPT	GO
`for (let i = 0; i < n; i++)`	`for i := 0; i < n; i++`
`while (cond)`	`for cond`
`while (true) { ... break; }`	`for { ... break }`
`for (const item of array)`	`for _, item := range slice`
`for (const key in object)`	`for key, value := range map`

Coding Challenges

CHALLENGE 1

Write a classic counting for loop that prints every even number from 0 to 10 (inclusive), using i += 2 in the post clause instead of i++.

View solution

CHALLENGE 2

Write a for loop used as a while loop: starting with balance := 100, keep subtracting 30 and printing the new balance each time, stopping (using the condition, not break) once balance would go below 0.

View solution

CHALLENGE 3

Given colours := []string{"red", "green", "blue", "yellow"}, use for...range with the blank identifier _ to print just the values, one per line, with no index shown.

View solution

CHAPTER 4 QUICK REFERENCE

- **for i := 0; i < n; i++ { }** — classic counting loop, like JavaScript's for
- **for condition { }** — Go's while loop; there is no separate while keyword
- **for { }** — infinite loop; use break to exit, continue to skip an iteration
- **for index, value := range slice { }** — like for...of, but always gives both index and value
- **for key, value := range map { }** — like for...in, conceptually; order is NOT guaranteed
- **_ (blank identifier)** — discards a value Go would otherwise require you to use
- **No ++i prefix form** — i++ and i-- are statements, always written after the variable
- **Next chapter:** functions — multiple return values, named returns, and variadic parameters

CHAPTER 5 OF 7

Functions: Multiple Returns, Named Returns, Variadic Params

COURSE 1 · CH 5

Functions: Multiple Returns, Named Returns, and Variadic Parameters

Go functions can hand back more than one value at once — the foundation of Go's entire error-handling style

Every function so far (`main`) has returned nothing. JavaScript functions return exactly one value — wrapping several values up means returning an array or object instead. Go does something JavaScript can't: a function can return **multiple, separate values** directly, no wrapping required. This single feature underpins almost every standard-library function in Go, including `fmt.Println`'s lesser-known cousin `fmt.Sscanf` and error handling throughout the language.

A Basic Function with Types on Both Ends

```
func add(a int, b int) int {
        return a + b
}

fmt.Println(add(3, 4))   // 7
```

Each parameter's type follows its name (`a int`), and the return type follows the closing parenthesis (`) int {`) — there's no `function` keyword equivalent confusion here since Go only has one function syntax, unlike JavaScript's three (Fundamentals Chapter 5).

SHARED TYPE SHORTHAND

When consecutive parameters share a type, the type only needs writing once: `func add(a, b int) int` means exactly the same as `func add(a int, b int) int`.

Multiple Return Values

```
func divide(a, b float64) (float64, string) {
        if b == 0 {
                return 0, "cannot divide by zero"
        }
        return a / b, ""
}

result, errMsg := divide(10, 2)
fmt.Println(result, errMsg)   // 5 ""
```

`(float64, string)` in the return position declares **two** separate return values — every `return` statement inside must supply both. The caller receives both at once with `result, errMsg := divide(10, 2)`, destructuring-style, but without any array or object ever being created.

The error Type — Go's Real Error-Handling Pattern

```
import (
        "errors"
        "fmt"
)

func divide(a, b float64) (float64, error) {
        if b == 0 {
                return 0, errors.New("cannot divide by zero")
        }
        return a / b, nil
}

result, err := divide(10, 0)
if err != nil {
        fmt.Println("Error:", err)
} else {
        fmt.Println("Result:", result)
}
```

This is the real, idiomatic Go pattern, used constantly throughout the language: a function returns its normal result *plus* an `error`, which is `nil` (Go's `null`) when nothing went wrong. `if err != nil` immediately after almost every call is so common it's practically a Go signature — this replaces the `try/catch` pattern from JavaScript Fundamentals Chapter 10 entirely; Go has no exceptions to throw or catch for ordinary error handling.

Named Return Values

```go
func rectangleStats(width, height float64) (area, perimeter float64) {
        area = width * height
        perimeter = 2 * (width + height)
        return   // "naked return" — returns area and perimeter automatically
}

a, p := rectangleStats(5, 3)
fmt.Println(a, p)   // 15 16
```

Naming the return values in the function signature (`area, perimeter float64`) pre-declares them as variables, usable directly inside the function body. A bare `return` with nothing after it — a **naked return** — automatically sends back whatever those named variables currently hold. This is mostly a readability tool for short functions; longer functions are usually clearer with an explicit `return area, perimeter`.

Variadic Parameters — Go's Version of Rest Parameters

```go
func sum(numbers ...int) int {
        total := 0
        for _, n := range numbers {
                total += n
        }
        return total
}

fmt.Println(sum(1, 2))         // 3
fmt.Println(sum(1, 2, 3, 4))   // 10
```

`...int` before the parameter type marks it as **variadic** — Go's direct equivalent of JavaScript's rest parameter (`...numbers`) from Intermediate Chapter 1. Inside the function, `numbers` behaves as a real slice, so `range` works on it exactly as it did in Chapter 4.

JAVASCRIPT	GO
Returns one value (or an array/object of several)	`func f() (int, string)` — returns several, separately
try/catch + throw	return value, error + if err != nil
function f(...args)	func f(args ...int)
No named return concept	func f() (result int) { ...; return }

Coding Challenges

CHALLENGE 1

Write a function divide(a, b float64) (float64, error) that returns an error from the errors package if b is 0, otherwise the division result and nil. Call it twice — once with b = 0, once with a real divisor — handling both with if err != nil.

View solution

CHALLENGE 2

Write a function minMax(numbers ...int) (min, max int) using named return values and a naked return, that finds the smallest and largest values among any number of arguments. Call it with at least 5 numbers.

View solution

CHALLENGE 3

Write a function describeNumber(n int) (string, bool) that returns "even" or "odd" as the first value, and whether n is positive as the second (boolean) value. Call it with three different numbers, printing both returned values each time.

View solution

CHAPTER 5 QUICK REFERENCE

- **func name(param type) returnType { }** — basic function shape
- **func name() (typeA, typeB) { return a, b }** — multiple return values, no wrapping needed
- **error type + nil** — Go's standard error pattern; check with if err != nil
- **errors.New("message")** — creates a basic error value
- **Named returns:** func f() (result type) — pre-declares the variable; "return" alone sends it back
- **Variadic parameters:** func f(args ...type) — Go's equivalent of JavaScript's rest parameter
- **No try/catch** — error handling is just normal values and normal if statements
- **Next chapter:** arrays and slices — Go's two array-like types, and why slices are used almost everywhere

CHAPTER 6 OF 7

Arrays and Slices

COURSE 1 · CH 6

Arrays and Slices

Go actually has two list-like types — but only one of them gets used in everyday code

Chapter 4 already used `[]string{...}` without fully explaining it. Go has both **arrays** (fixed-size, rarely used directly) and **slices** (flexible, growable — the real workhorse, and the closest equivalent to a JavaScript array). This chapter covers both, but spends most of its time on slices, since that's what real Go code uses almost everywhere.

Arrays — Fixed Size, Part of the Type Itself

```
var scores [3]int = [3]int{88, 72, 95}

fmt.Println(scores)        // [88 72 95]
fmt.Println(scores[0])     // 88
fmt.Println(len(scores))   // 3
```

`[3]int` means "an array of exactly 3 ints" — the size is part of the type itself, fixed forever once declared. `[3]int` and `[4]int` are considered entirely different types; an array can never grow or shrink. This rigidity is exactly why slices exist and are used almost everywhere instead.

Slices — The Type You'll Actually Use

```
fruits := []string{"apple", "banana", "cherry"}

fmt.Println(fruits[0])         // "apple"
fmt.Println(len(fruits))     // 3
```

`[]string` — square brackets with **no number** inside — declares a slice, not an array. That missing number is the entire visual difference between the two, and it matters enormously: a slice can grow after creation, which an array never can.

Growing a Slice with append

```
numbers := []int{1, 2, 3}

numbers = append(numbers, 4)
fmt.Println(numbers)   // [1 2 3 4]

numbers = append(numbers, 5, 6)
fmt.Println(numbers)   // [1 2 3 4 5 6]
```

`append` is the direct equivalent of JavaScript's `push` (Fundamentals Chapter 6) — with one crucial difference: `append` **returns** a new slice rather than modifying in place, so the result must always be reassigned (`numbers = append(numbers, 4)`). Forgetting the reassignment is a common mistake — the original variable simply won't reflect the new element.

> **A SLICE IS A "VIEW" ONTO AN UNDERLYING ARRAY**
>
> Behind the scenes, a slice tracks a pointer, a length, and a capacity into a real array Go manages automatically. This is normally invisible day-to-day, but it explains why `append` sometimes allocates a brand-new underlying array (when capacity runs out) and sometimes doesn't — either way, always use the value `append` returns.

Slicing a Slice — The [start:end] Syntax

```
letters := []string{"a", "b", "c", "d", "e"}

fmt.Println(letters[1:3])   // [b c] — index 1 up to (not including) index 3
fmt.Println(letters[:2])    // [a b] — from the start, up to index 2
fmt.Println(letters[3:])    // [d e] — from index 3 to the end
```

`letters[1:3]` extracts a sub-slice, similar in spirit to JavaScript's array `.slice()` method, but built directly into Go's bracket syntax rather than a method call. The end index is always exclusive, the same convention as JavaScript's `slice()`.

Iterating and Combining with Earlier Chapters

```go
prices := []float64{9.99, 14.50, 3.25}

total := 0.0
for _, price := range prices {
        total += price
}

fmt.Printf("Total: %.2f\n", total)   // Total: 27.74
```

Combining Chapter 4's `range` with a running total is Go's version of JavaScript's `reduce` — there's no built-in `reduce` method on slices in Go, so this manual loop pattern is the idiomatic replacement. `%.2f` in `Printf` formats a float to exactly 2 decimal places.

JAVASCRIPT	GO
`const arr = [1, 2, 3];`	`arr := []int{1, 2, 3} (slice)`
`arr.push(4)`	`arr = append(arr, 4)`
`arr.length`	`len(arr)`
`arr.slice(1, 3)`	`arr[1:3]`
`arr.reduce((sum, x) => sum + x, 0)`	`manual for...range loop with a running total`

Coding Challenges

CHALLENGE 1

Create a slice named temps containing the floats 18.5, 22.0, 15.5, then use append to add 30.0 and 27.5 (in one call). Print the final slice and its length using len().

View solution

CHALLENGE 2

Create a slice named words containing 6 short strings of your choice. Print the first 3 using slicing syntax, then print the last 3 using slicing syntax, without using any literal index numbers higher than what's needed for each.

View solution

CHALLENGE 3

Create a slice of ints called scores with at least 5 values. Write a loop that calculates both the total and the average (total / number of scores, as a proper float64 result), printing both using Printf with %.2f.

View solution

CHAPTER 6 QUICK REFERENCE

- **[N]Type** — a fixed-size array; size is part of the type, never grows
- **[]Type** — a slice; flexible, growable, the type used almost everywhere in real Go code
- **append(slice, value)** — adds an element; ALWAYS reassign the result back
- **len(slice)** — number of elements, works on arrays, slices, strings, and maps
- **slice[start:end]** — sub-slice; end index is exclusive, either side can be omitted
- **No built-in map/filter/reduce** — manual for...range loops are the idiomatic replacement
- **%.2f** — Printf verb for a float formatted to 2 decimal places
- **Next chapter:** maps — Go's key/value type, the rough equivalent of a JavaScript object

CHAPTER 7 OF 7

Maps

COURSE 1 · CH 7

Maps: Go's Key/Value Type

The rough equivalent of a JavaScript object — but with one type for every key and one type for every value

Chapter 4 briefly used `map[string]int{...}` while explaining `range`. A Go **map** is the closest equivalent to a JavaScript object's key/value behaviour (Fundamentals Chapter 7) — except every key must be the same type, and every value must be the same type, declared up front as part of the map's own type.

Creating and Reading a Map

```
scores := map[string]int{
        "maths":   88,
        "science": 72,
        "art":      95,
}

fmt.Println(scores["maths"])   // 88
```

`map[string]int` reads as "a map from `string` keys to `int` values." Unlike JavaScript, where an object can hold a string here and a number there, every value in this map must be an `int` — trying to add a string value would be a compile error.

Adding and Updating Entries

```
scores["history"] = 79   // adds a new key
scores["maths"] = 90     // updates an existing key
```

Adding and updating use the same bracket-assignment syntax — Go decides which happened based on whether the key already existed, the same way JavaScript object property assignment works.

The Comma-Ok Idiom — Checking If a Key Exists

```
value, exists := scores["french"]

if exists {
        fmt.Println("Score:", value)
} else {
        fmt.Println("No score recorded for french")
}
```

Reading a missing key from a map doesn't error — it silently returns the value type's zero value (Chapter 2), which can hide a real bug. `value, exists := scores["french"]` — reading TWO things from a single map lookup — is how Go distinguishes "the key exists and its value happens to be 0" from "the key genuinely doesn't exist." This pattern is called **comma-ok**, and appears constantly in real Go code.

> **A MISSING KEY LOOKS IDENTICAL TO A REAL ZERO VALUE**
>
> `scores["french"]` alone (without comma-ok) returns `0` whether or not "french" was ever added — there's no way to tell the difference from that single value alone. Always use the two-value form when "did this exist at all?" actually matters.

Deleting a Key

```
delete(scores, "art")
fmt.Println(scores)   // "art" is gone entirely
```

`delete(map, key)` is a built-in function (not a method) — the direct equivalent of JavaScript's `delete person.isStudent` from Fundamentals Chapter 7.

Looking Ahead: structs (Briefly)

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

p := Person{Name: "Philip", Age: 35}
fmt.Println(p.Name)   // "Philip"
```

Maps are excellent for "any number of similarly-shaped entries" (a dictionary, a lookup table), but real-world records with a *fixed* set of named fields — a person, a product, an order — usually use a **struct** instead, Go's actual replacement for a JavaScript object literal. Structs get their own full treatment in Intermediate; this

preview just establishes that `p.Name` uses the same dot notation as Chapter 7's JavaScript objects, even though the underlying type is completely different from a map.

JAVASCRIPT	GO
`{ key: value }`	`map[string]int{"key": value}`
`obj.key = value`	`m[key] = value`
`delete obj.key`	`delete(m, key)`
`obj.key === undefined to check existence`	`value, exists := m[key] (comma-ok)`
`Mixed value types allowed`	`All values must be the same declared type`
`Fixed-shape record`	`struct (preview here, full coverage in Intermediate)`

Coding Challenges

CHALLENGE 1

Create a map stock := map[string]int{"apples": 10, "bananas": 5}. Add "oranges": 8 to it, update "apples" to 15, then print the whole map.

View solution

CHALLENGE 2

Using the same stock map, use the comma-ok idiom to check for "grapes" (which doesn't exist) and "apples" (which does), printing an appropriate message for each case.

View solution

CHALLENGE 3

Create a map inventory of at least 4 items with int quantities. Use a for...range loop to print each item with its quantity, and keep a running total using a plain variable, printing the total at the end. Then delete one item and print the map again to confirm it's gone.

View solution

CHAPTER 7 QUICK REFERENCE

- **map[KeyType]ValueType{...}** — every key shares one type, every value shares another
- **m[key] = value** — adds a new key or updates an existing one
- **value, exists := m[key]** — comma-ok idiom; the only reliable way to check if a key exists
- **delete(m, key)** — a built-in function, not a method
- **for key, value := range m { }** — order is NOT guaranteed (Chapter 4)
- **type Name struct { Field type }** — Go's fixed-shape record type, previewed here
- **Maps:** good for dynamic/lookup-style data. **Structs:** good for fixed-shape records
- **This completes Go Fundamentals.** Intermediate begins with structs in full, methods, and pointers.