Lesson 3: Data Types

Strings, Numbers, Arrays, Slices, Maps & Pointers

Overview: Go's Concrete Data Types

Go has a relatively small set of built-in data types compared to TypeScript, but they behave very differently under the hood. Understanding these types deeply is critical because Go doesn't hide memory details the way JS/TS does.

Category	Go Types	TS Equivalent
Integers	`int, int8, int16, int32, int64` `uint, uint8, uint16, uint32, uint64`	`number`
Floats	`float32, float64`	`number`
Strings	`string (immutable, UTF-8 bytes)`	`string`
Booleans	`bool`	`boolean`
Arrays	`[N]T  (fixed size, value type)`	`readonly tuple`
Slices	`[]T  (dynamic, reference type)`	`Array<T> / T[]`
Maps	`map[K]V`	`Map<K,V> / Record<K,V>`
Pointers	`*T  (address of a value)`	`N/A (no equivalent)`
Byte / Rune	`byte (uint8), rune (int32)`	`N/A (char-like)`

Key difference from TS: In TypeScript, `number` covers everything. In Go, you choose the exact size and signedness. This gives you precise control over memory and performance.

Numbers

Integer Types

```
var a int     = 42        // platform-dependent: 32 or 64 bit
var b int8    = 127       // -128 to 127
var c int32   = 2147483647 // ~2 billion
var d int64   = 9223372036854775807
var e uint8   = 255       // unsigned, 0 to 255 (same as byte)
```

In practice, just use `int` for most things. Use sized types when working with binary protocols, databases, or performance-critical code.

Float Types

```
var pi float64 = 3.14159265358979  // default, use this
var fast float32 = 3.14            // less precision, less memory

// Type conversion (Go does NOT auto-convert):
x := 10          // int
y := 3.5         // float64
z := float64(x) + y  // must explicitly convert int to float64
```

No implicit conversions! In TS, `10 + 3.5` just works. In Go, mixing int and float64 is a compiler error. You must convert explicitly.

Strings

Go strings are **immutable sequences of bytes**, always UTF-8 encoded.

```
name := "Alice"               // string literal
multiline := `This is a
raw string literal`            // backticks, no interpolation

len(name)                    // 5 (byte count, NOT character count!)
name[0]                      // 65 (byte value of 'A', NOT a string)
name + " Smith"               // concatenation
fmt.Sprintf("Hi %s, age %d", name, 30) // string interpolation
```

byte vs rune

```
// byte = uint8, a single byte
// rune = int32, a single Unicode code point (like a char)

s := "Hello, ■■"
len(s)                // 13 bytes (Chinese chars = 3 bytes each)
[]rune(s)             // len = 9 characters

for i, ch := range s {  // ch is a rune, iterates correctly
    fmt.Printf("%d: %c\n", i, ch)
}
```

Common string operations (strings package)

```
import "strings"

strings.Contains("hello", "ell")    // true
strings.HasPrefix("hello", "he")    // true
strings.ToUpper("hello")           // "HELLO"
strings.Split("a,b,c", ",")        // []string{"a", "b", "c"}
strings.Join([]string{"a","b"}, "-") // "a-b"
strings.ReplaceAll("foo", "o", "0")  // "f00"
strings.TrimSpace(" hi ")          // "hi"
```

Arrays (Fixed Size, Rarely Used Directly)

In Go, arrays have a **fixed size** that is part of the type. `[3]int` and `[5]int` are **completely different types**. Arrays are **value types** — assigning or passing makes a full copy. You'll rarely use arrays directly; slices are almost always what you want.

```
var nums [5]int                // [0, 0, 0, 0, 0]
names := [3]string{"a", "b", "c"} // initialized
auto := [...]int{1, 2, 3, 4}     // compiler counts: [4]int

// Arrays are VALUE TYPES (copied on assignment!)
a := [3]int{1, 2, 3}
b := a         // b is a COPY of a
b[0] = 99      // a[0] is still 1 (not affected!)
```

In TS, `const b = a` makes both point to the same data. In Go, arrays are **fully copied**. This catches many people off guard.

Slices (Dynamic — This Is What You'll Actually Use)

Slices are Go's answer to JavaScript arrays. They are **dynamic**, **reference-based**, and the collection type you'll use 95% of the time. A slice is a lightweight struct: a **pointer** to an underlying array, a **length**, and a **capacity**.

Creating Slices

```
// Slice literal (most common)
names := []string{"Alice", "Bob", "Charlie"}  // no size in brackets!

// Using make (pre-allocate length and capacity)
scores := make([]int, 5)      // len=5, cap=5, all zeros
buffer := make([]byte, 0, 100) // len=0, cap=100 (pre-allocated)

// Nil slice (valid, zero value)
var empty []int              // nil, len=0, cap=0
// nil slices work with append, len, range — no nil checks needed
```

Appending, Slicing, and Iterating

```
// Append (like JS .push() but returns a new slice)
names = append(names, "Diana")          // add one
names = append(names, "Eve", "Frank")    // add multiple
all := append(names, moreNames...)      // spread another slice

// Slicing (like JS .slice())
first3 := names[0:3]    // elements 0, 1, 2
from2 := names[2:]      // from index 2 to end
upTo3 := names[:3]      // from start to index 3 (exclusive)

// Iterating
for i, name := range names {
    fmt.Printf("%d: %s\n", i, name)
}
for _, name := range names {  // ignore index
    fmt.Println(name)
}
```

CRITICAL: append() returns a new slice! You must always reassign: `names = append(names, "new")`. Just calling append without capturing the return does nothing. This is the #1 Go slice mistake.

Length vs Capacity

```
s := make([]int, 3, 10)  // len=3, cap=10
len(s)                   // 3 (current elements)
cap(s)                   // 10 (room before reallocation)
// When you append beyond capacity, Go allocates a new larger array
// and copies elements over. Capacity typically doubles each time.
```

Common Slice Operations

```
// Delete element at index i (order preserved)
s = append(s[:i], s[i+1:]...)

// Copy a slice (avoid sharing the underlying array)
dst := make([]int, len(src))
copy(dst, src)

// Since Go 1.21: slices package helpers
import "slices"
slices.Contains(names, "Alice")  // true
slices.Sort(nums)               // sorts in-place
```

Maps (Key-Value Pairs)

Maps are Go's hash tables — like TS `Map<K, V>` or plain objects. Keys must be **comparable types** (strings, ints, structs). Slices, maps, and functions cannot be keys.

Creating and Using Maps

```
// Map literal
ages := map[string]int{
    "Alice": 30,
    "Bob":   25,
}

// Using make
scores := make(map[string]int)   // empty map, ready to use

// CRUD operations
ages["Charlie"] = 35            // create / update
age := ages["Alice"]            // read (returns 30)
delete(ages, "Bob")             // delete
len(ages)                       // 2
```

Checking if a Key Exists (The Comma-Ok Idiom)

```
// Reading a missing key returns the zero value (0, "", false, nil)
// How do you tell "key doesn't exist" from "value is 0"?

age, ok := ages["Diana"]
if ok {
    fmt.Println("Found:", age)
} else {
    fmt.Println("Not found")
}

// Shorter (idiomatic):
if age, ok := ages["Diana"]; ok {
    fmt.Println(age)
}
```

The comma-ok idiom is used everywhere in Go — maps, type assertions, channel receives. The second return value `ok` is a bool telling you if the operation succeeded.

Iterating Over Maps

```
for name, age := range ages {
    fmt.Printf("%s is %d\n", name, age)
}
for name := range ages {  // just keys
    fmt.Println(name)
}
```

Warning: Map iteration order is **random** in Go! Unlike JS objects (which preserve insertion order), Go deliberately randomizes iteration. Sort keys into a slice first if order matters.

Map Gotchas

```
// A nil map can be READ but NOT WRITTEN to
var m map[string]int   // nil map
_ = m["key"]          // ok, returns 0
m["key"] = 1          // PANIC! assignment to nil map

// Always initialize before writing:
```

```
m = make(map[string]int)  // now safe
```

Pointers

Pointers are probably the biggest conceptual gap from TS/JS, since JavaScript has no pointers. A pointer holds the **memory address** of a value.

The Basics: & and *

```
x := 42
p := &x       // p is a *int (pointer to int), holds address of x
fmt.Println(p)  // 0xc0000b2008 (memory address)
fmt.Println(*p) // 42 (dereference: get value at that address)

*p = 100        // change the value through the pointer
fmt.Println(x)  // 100 (x was modified!)
```

`&` = "give me the address of" `*` = "give me the value at this address" `*int` = the type "pointer to int"

Why Pointers Matter: Value vs Reference Semantics

```
// WITHOUT pointer: function gets a COPY, original unchanged
func double(n int) {
    n = n * 2   // modifies the copy only
}
x := 5
double(x)
fmt.Println(x)  // still 5!

// WITH pointer: function can modify the original
func double(n *int) {
    *n = *n * 2 // modifies the value at the address
}
x := 5
double(&x)      // pass the address
fmt.Println(x)  // 10!
```

Pointers with Structs (The Most Common Use Case)

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

// Return a pointer to avoid copying a large struct
func NewUser(name string, age int) *User {
    return &User{Name: name, Age: age}
}

u := NewUser("Alice", 30)
u.Name            // "Alice" (auto-dereferences, no -> needed)
u.Age = 31        // modifies the original struct
```

When to use pointers: (1) When you need to modify a value in a function. (2) When passing large structs (avoids copies). (3) When you need to represent "nothing" (nil pointer, like null in TS). (4) Almost always for method receivers on structs.

Pointer Safety: Go vs C

Unlike C, Go has **no pointer arithmetic**. You can't increment a pointer or do math with addresses. Go's garbage collector means you don't manually free memory. The main risk is **nil pointer dereference**:

```
var p *User        // nil pointer
p.Name             // PANIC: nil pointer dereference

// Always check for nil:
if p != nil {
    fmt.Println(p.Name)
}
```

Data Types Quick Reference: Go vs TypeScript

Operation	TypeScript	Go
Create list	`const a = [1, 2, 3]`	`a := []int{1, 2, 3}`
Add to end	`a.push(4)`	`a = append(a, 4)`
Get length	`a.length`	`len(a)`
Slice/subset	`a.slice(1, 3)`	`a[1:3]`

Spread/concat	`[...a, ...b]`	`append(a, b...)`
Check includes	`a.includes(x)`	`slices.Contains(a, x)`
Map/filter	`a.map(fn) / a.filter(fn)`	`for loop (manual)`
Create map	`{ key: value }`	`map[string]int{"key": val}`
Access key	`obj.key or obj["key"]`	`m["key"]`
Delete key	`delete obj.key`	`delete(m, "key")`
Key exists?	`"key" in obj`	`_, ok := m["key"]`
String template	`` `Hello ${name}` ``	`fmt.Sprintf("Hello %s", name)`
String split	`s.split(",")`	`strings.Split(s, ",")`
Null check	`if (x !== null)`	`if x != nil`
Pass by ref	`automatic for objects`	`explicit with &x, *T`

Key Takeaways

- **Go has no generic "number" type** — you choose int, int64, float64, etc. No implicit conversions between them.
- **Strings are byte slices** — len() returns bytes, not characters. Use rune for Unicode-aware operations.
- **Arrays are fixed-size value types** — you'll almost never use them directly. Use slices instead.
- **Slices are your main collection** — dynamic, reference-based. Always reassign append(): `s = append(s, x)`.
- **Maps must be initialized** before writing — a nil map panics on write. Always use make() or a literal.
- **Map iteration order is random** — unlike JS objects. Sort keys explicitly if order matters.
- **Pointers give explicit control** over pass-by-value vs pass-by-reference. Use & to get address, * to dereference.
- **No map/filter/reduce** — you write for loops. The slices package (Go 1.21+) adds some helpers.
- **The comma-ok idiom** (`val, ok := m[key]`) is used everywhere for safe access.

Next Lesson: Lesson 4 will cover Go's type system, structs, interfaces, methods, and control flow — how you define and compose your own types.