Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 7

Maps and Slices

If you are coming from Java, two data structures will carry the weight of almost every program you write in Go: maps and slices. Maps replace `HashMap<K,V>` — same O(1) lookup, far less ceremony. Slices replace `ArrayList<E>` — same dynamic growth, but backed by a plain array and built directly into the language. Without a solid grasp of these two, simple things turn painful: you cannot iterate a collection in deterministic order, you write a value into a map and the program panics, or you slice off "just the first two elements" and silently corrupt the original. A Java programmer expects `ArrayList` to be a self-contained object and `HashMap` to hide its storage; Go exposes the seams, and the aliasing behavior that follows from those exposed seams is the single biggest source of surprise bugs in early Go code. Getting maps and slices right up front saves you from chasing those bugs in every later chapter. This chapter covers maps first (key–value lookup), then slices (dynamic sequences), because maps tend to surprise Java programmers more and are worth getting right before the subtleties of slice aliasing land on top.

Maps

A map in Go is an unordered collection of key–value pairs. The Java equivalent is `HashMap<K,V>`. Like `HashMap`, Go maps offer O(1) average-case lookup, insertion, and deletion. Unlike `HashMap`, Go maps are a built-in type with dedicated syntax.

Map Type and Declaration

The type of a map with keys of type `K` and values of type `V` is written `map[K]V`.

```
var plays map[string]int // nil map --- not yet initialized
```

`K` must be **comparable**: any type that supports `==` and `!=`. Strings, integers, floats, booleans, pointers, channels, arrays of comparable elements, and structs whose fields are all comparable are valid key types. Slices, maps, and functions are **not** comparable and cannot be used as keys.

Wut: Java lets you use any object as a `HashMap` key as long as it implements `hashCode()` and `equals()`. Go requires the key type to support `==` at the language level. You cannot use a `[]byte` as a Go map key; convert it to `string` first.

There are two ways to create a non-nil map.

Map literal:

```
streams := map[string]int{
    "Saltwater":       1_200_000_000, // Chicane
    "Out Of The Blue": 980_000_000,   // System F
```

```
    "Gamemaster":      750_000_000,   // Matt Darey & Lost Tribe
}
```

make:

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

`make(map[K]V)` returns an initialized, empty map. You can also pass a size hint — `make(map[K]V, 100)` — which pre-allocates internal buckets but does not set a maximum size.

Use `make` when you need an empty map, possibly pre-sized. Use a literal when you have initial values — literals also compose naturally for nested maps:

```
grades := map[string]map[string]int{
    "CMPE 30": {
        "Ben":  22,
        "Amy":  88,
        "Fred": 32,
    },
    "CMPE 50": {
        "Qi":  90,
        "Cal": 102,
    },
}
```

Trap: A `var` declaration without an initializer gives you a **nil map**. Reading from a nil map is safe and returns the zero value. **Writing to a nil map panics at runtime.**

```
var m map[string]int
fmt.Println(m["key"]) // fine --- prints 0
m["key"] = 1          // panic: assignment to entry in nil map
```

Always initialize with a literal or `make` before writing.

Map Operations

The four basic operations on a map are read, write, delete, and count.

```
m := map[string]int{
    "cumbia":    92,
    "reggaeton": 98,
    "afrobeats": 120,
}

fmt.Println(m["cumbia"])   // 92       --- read
m["drill"] = 140           //          --- write
delete(m, "reggaeton")     //          --- delete
fmt.Println(len(m))        // 3        --- count: cumbia, afrobeats, drill
```

The signatures of the relevant built-in functions are:

```
func delete(m map[K]V, k K)  // remove the entry with key k; no-op if k is absent
func len(v Type) int         // number of entries in a map (or length of slice/string/channel)
```

Reading a key that is not in the map does **not** panic or throw. It returns the zero value for the value type:

```
bpm := m["classical"] // 0 --- key not present, zero value returned
```

This is safe but can hide bugs: you cannot tell whether `bpm` is `0` because classical music has no tempo entry, or because someone explicitly stored `0`. The comma-ok idiom (next section) resolves that ambiguity. [*no-in-band-errors*]

The Comma-ok Idiom

To test whether a key is actually present in a map, use the two-value assignment form:

```
v, ok := m[k] // v is the value (or zero), ok is true only if k exists
```

`ok` is a `bool`. It is `true` if the key was found, `false` if it was not. This is idiomatic Go; you will see it everywhere.

```
catalog := map[string]string{
    "Emerald Triangle 2012": "Angoscia",
    "Better Off Alone":      "DJ Cobra",
    "Children":              "Robert Dream House",
}

if artist, ok := catalog["Better Off Alone"]; ok {
    fmt.Println("Found:", artist) // Found: DJ Cobra
} else {
    fmt.Println("Not in catalog")
}
```

Tip: The Java idiom is `if (map.containsKey(k)) { v = map.get(k); }` — two lookups. The Go comma-ok form is a single lookup that returns both the value and the presence flag. Prefer the comma-ok form over comparing the value to its zero value; the zero value might be a legitimate stored value.

Iteration Order

Ranging over a map with `for range` visits every key–value pair, but **in a random order that changes on every iteration** — even two range loops over the same map in the same run can visit keys in different orders.

```
bpm := map[string]int{
    "amapiano": 112,
    "hyperpop": 160,
    "lo-fi":    85,
}

for genre, b := range bpm {
    fmt.Printf("%s: %d\n", genre, b) // order varies every run
}
```

This is a deliberate design choice. Go randomizes map iteration to make it immediately obvious if your code accidentally depends on order. Java's `HashMap` also makes no ordering guarantee, but it tends to be stable in practice within a single run, which can mask order dependencies.

Trap: Never depend on map iteration order. If you need sorted output, collect the keys into a slice and sort it first. Add `"sort"` to your imports at the top of the file:

```
import "sort"
```

Then collect and sort the keys:

```
keys := make([]string, 0, len(bpm))
for k := range bpm {
    keys = append(keys, k)
}
sort.Strings(keys)

for _, k := range keys {
    fmt.Printf("%s: %d\n", k, bpm[k])
}
```

Clearing a Map

Since Go 1.21, the `clear` built-in deletes all entries from a map:

```
func clear(m map[K]V) // delete all entries; map remains non-nil and usable
```

```
m := map[string]int{"a": 1, "b": 2, "c": 3}
clear(m)
fmt.Println(len(m)) // 0
m["d"] = 4          // fine --- map is still usable
```

After `clear`, the map is empty but not nil. This is different from assigning `m = make(map[K]V)`, which allocates a new map and abandons the old one. `clear` reuses the existing map, which can be useful when the map is shared or when you want to avoid a reallocation.

Sets

Go has no built-in set type — there is no `HashSet` waiting in the standard library the way there is in Java's `java.util`. The idiomatic substitute is a map whose keys are the set's elements and whose values carry no information. Two encodings are common: `map[T]bool` and `map[T]struct{}`.

The empty struct, `struct{}`, is the idiomatic value type because it occupies **zero bytes** — you are using the map purely for its keys, so the value should cost nothing:

```
seen := map[string]struct{}{}  // a set of strings

seen["reggaeton"] = struct{}{} // add an element
seen["dembow"] = struct{}{}

_, ok := seen["reggaeton"]     // membership test (the comma-ok idiom)
fmt.Println(ok)                // true

delete(seen, "dembow")         // remove an element
fmt.Println(len(seen))         // 1 --- the cardinality of the set

for genre := range seen {      // iterate the elements (random order)
    fmt.Println(genre)
}
```

Every set operation is just a map operation you have already seen: add with `set[x] = struct{}{}`, test membership with the comma-ok idiom, remove with `delete`, and count with `len`. The doubled-up `struct{}{}`

reads as "a value of the empty-struct type" — the inner `struct{}` is the type, and the trailing `{}` is a literal of that type.

Tip: `map[T]bool` is the friendlier-looking alternative: you add with `set[x] = true` and test with `if set[x]`, leaning on the `false` zero value for absent keys. It costs one byte per entry instead of zero, which rarely matters. Reach for `map[T]struct{}` when the set is large or memory-sensitive, or when you want the type itself to announce "the values here are meaningless." Use `map[T]bool` when readability wins.

Trap: With the `map[T]bool` encoding, do not confuse "absent" with "present but `false`." Testing `set[x]` returns `false` both when `x` was never added and when you deliberately stored `false`. If `false` is ever a value you might store, use the comma-ok form (`_, ok := set[x]`) or switch to `map[T]struct{}`, where the only way a key can exist is that you put it there.

Arrays

Go has arrays, but you will rarely use them directly. An array in Go is a **value type** with a **fixed size that is part of the type itself**.

```
var a [3]int            // [0 0 0]    --- zero-initialized
b := [3]int{10, 20, 30} // [10 20 30]
c := [...]int{1, 2, 3}  // [1 2 3]    --- compiler counts the elements
```

The `[...]` form lets the compiler infer the length from the literal. After that point the length is still fixed — `c` is type `[3]int`.

Array Length Is Part of the Type

`[3]int` and `[4]int` are **different types**. You cannot pass a `[3]int` to a function that expects a `[4]int`, and you cannot assign one to the other.

```
var x [3]int
var y [4]int
// x = y  // compile error: cannot use [4]int as [3]int
```

Wut: In Java, `int[]` is a reference type regardless of length. In Go, `[3]int` and `[4]int` are as distinct as `int` and `string`. This is why arrays are rarely used as function parameters — you would have to hard-code the length into every function signature. Slices solve this problem.

Arrays Are Value Types

Assigning an array copies every element. Passing an array to a function passes a full copy.

```
a := [3]int{1, 2, 3}
b := a      // b is an independent copy
b[0] = 99
fmt.Println(a[0]) // 1  --- a is unchanged
fmt.Println(b[0]) // 99
```

Java arrays are reference types: assigning `int[] b = a` makes both variables point at the same array. Go arrays have value semantics: every assignment is a copy.

Slices

A slice is a **three-field descriptor** that points into a contiguous region of an underlying array:

Field	Meaning
pointer	address of the first element visible through the slice
length	number of elements currently accessible
capacity	number of elements from the pointer to the end of the backing array

You can think of it as:

```
// conceptual --- not real Go syntax
type sliceHeader struct {
    ptr *T   // pointer to backing array
    len int  // number of accessible elements
    cap int  // elements from ptr to end of backing array
}
```

The type `[]int` (square brackets with no number) is a slice of `int`. `[3]int` (with a number) is an array of three `int` values. The presence or absence of the number is everything.

`len` and `cap` are built-in functions:

```
func len(v Type) int // number of elements in v (slice, array, map, string, channel)
func cap(v Type) int // capacity of v (slice or array; array cap == array len)
```

```
s := []int{10, 20, 30, 40, 50}
fmt.Println(len(s)) // 5
fmt.Println(cap(s)) // 5
```

Slice Literals and `make`

Slice Literal

A slice literal is written just like an array literal, but without a length:

```
s := []int{1, 2, 3}  // len=3, cap=3
```

make

`make` allocates a backing array and returns a slice header pointing to it:

```
func make(t Type, size ...int) Type // allocate and initialize a slice, map, or channel
```

For slices, the two common forms are:

```
s1 := make([]int, 5)     // len=5, cap=5    --- five zeros
s2 := make([]int, 0, 10) // len=0, cap=10   --- empty but room for 10 elements
```

`make([]int, 0, 10)` is the idiomatic way to pre-allocate when you know roughly how many elements you will append. It avoids repeated reallocation as the slice grows. This mirrors Java's `new ArrayList<>(initialCapacity)`: you are not filling the slice, you are reserving room so that `append` calls don't trigger repeated reallocations.

```
s := make([]int, 0, 100) // empty slice, room for 100 elements
for i := range 100 {
    s = append(s, i)      // no reallocation needed
}
```

Tip: `make([]T, 0, n)` is the idiomatic way to pre-allocate a slice you will fill with `append`. `make([]T, n)` is for when you want `n` zero-valued elements you will assign by index.

`nil` Slice vs Empty Slice

```
var s []int          // nil slice   --- pointer is nil, len=0, cap=0
e := []int{}         // empty slice --- pointer is non-nil, len=0, cap=0
m := make([]int, 0)  // empty slice --- same as above
```

Both a nil slice and an empty slice have `len` of zero and behave identically with `range`, `append`, and `len`. The only difference is `s == nil`, which is `true` for a nil slice and `false` for an empty slice. APIs should not distinguish between the two; callers should not need to care whether they received a nil slice or an empty one. [*no-nil-vs-empty-api*]

Tip: A nil slice is a perfectly valid starting point. You can `append` to a nil slice without any initialization. Prefer `var s []int` over `s := []int{}` when you are building a slice with `append`; it is slightly more idiomatic and avoids an unnecessary allocation. [*nil-slice-preferred*]

`append`

`append` is the built-in function for adding elements to a slice:

```
func append[T any](s []T, elems ...T) []T // append elems to s and return the new slice
```

`append` always returns a new slice header. You must use the returned value — ignoring it discards the length update.

```
s := []int{1, 2, 3}
s = append(s, 4)       // s is now [1 2 3 4]
s = append(s, 5, 6, 7) // append multiple elements at once
```

The Backing Array May Be Replaced

When there is room in the backing array (i.e., `len(s) < cap(s)`), `append` extends the slice in place and no new allocation happens. When the backing array is full, `append` allocates a larger array, copies the existing elements, and returns a slice pointing at the new array.

```
s := make([]int, 0, 3) // len=0, cap=3 --- room for 3 before reallocation
s = append(s, 1)       // len=1, cap=3 --- no reallocation
s = append(s, 2)       // len=2, cap=3 --- no reallocation
s = append(s, 3)       // len=3, cap=3 --- no reallocation
s = append(s, 4)       // len=4, cap>=6 --- new backing array allocated and old contents copied
```

The growth factor is not guaranteed by the specification, but the standard library currently roughly doubles capacity for small slices, then grows more slowly for large ones. The exact strategy can change between Go releases.

Trap: Because `append` may return a slice backed by a **new** array, any other slice that was sharing the old backing array is **not updated**. Always reassign: `s = append(s, elem)`. Never write `append(s, elem)` without capturing the return value — the compiler will reject it anyway with "evaluated but not used".

Appending a Slice

To append all elements from one slice onto another, use the `...` spread operator:

```
a := []int{1, 2, 3}
b := []int{4, 5, 6}
a = append(a, b...) // a is now [1 2 3 4 5 6]
```

The `b...` syntax unpacks `b` into individual arguments. Without it, `append(a, b)` would be a type error because `b` is a `[]int`, not an `int`.

`copy`

`copy` copies elements from one slice to another:

```
func copy(dst, src []Type) int // copy min(len(dst),len(src)) elements; return count copied
```

`copy` copies exactly `min(len(dst), len(src))` elements — it never grows `dst`. It does not append; the destination must already have enough length.

```
src := []int{1, 2, 3, 4, 5}
dst := make([]int, 3)   // len=3
n := copy(dst, src)     // copies 3 elements (limited by dst length)
fmt.Println(dst)        // [1 2 3]
fmt.Println(n)          // 3
```

The primary use of `copy` is to **break aliasing**: when you need an independent slice that will not affect the original if you modify it.

```
original := []int{10, 20, 30}
clone := make([]int, len(original))
copy(clone, original)
clone[0] = 99
fmt.Println(original[0]) // 10 --- original is untouched
fmt.Println(clone[0])    // 99
```

Slicing Expressions

You can derive a new slice from an existing slice (or array) using a **slicing expression**. The result shares the backing array with the original.

Two-Index Form

```
s[low:high] // elements from index low up to (not including) high
```

```
s := []string{"a", "b", "c", "d", "e"}
t := s[1:3] // ["b" "c"]   --- len=2, cap=4 (from index 1 to end of backing array)
```

Either index may be omitted; the defaults are `0` and `len(s)`:

```
s[:3]  // same as s[0:3]
s[2:]  // same as s[2:len(s)]
s[:]   // same as s[0:len(s)]  --- a slice of the whole thing
```

Three-Index Form

The three-index form `s[low:high:max]` sets the **capacity** of the new slice as well as its length.

```
s := []int{10, 20, 30, 40, 50}
t := s[1:3:3] // len=2, cap=2  --- elements [20 30]; cannot reach [40 50]
```

Without the third index, `t` would have `cap=4` and an `append` to `t` could overwrite `s[3]`. With `s[1:3:3]`, the capacity equals the length, so any `append` to `t` forces a new backing array and never touches `s`.

Tip: Use the three-index form when you return a sub-slice from a function and want to guarantee that the caller cannot accidentally modify the original slice by appending to the returned sub-slice.

Slice Aliasing

Re-slicing does **not** copy data. Both the original and the new slice point into the same backing array. Modifying elements through one slice modifies what the other sees.

```
track := []string{"Flaming June", "Sandstorm", "Gouryella", "The Sound of Silence"}
top2 := track[:2]         // shares backing array
top2[0] = "Crazy Train"   // modifies the backing array
fmt.Println(track[0])     // Crazy Train  --- track sees the change too
```

This is efficient (no copying) but surprising when you forget about it.

```
a := []int{1, 2, 3, 4, 5}
b := a[1:3] // b is [2 3], cap=4
b = append(b, 99)
fmt.Println(a) // [1 2 3 99 5]  --- append overwrote a[3]!
```

Here `b` had capacity for two more elements before reaching the end of `a`'s backing array. The `append` wrote `99` into index 3 of the backing array, which is `a[3]`.

Trap: Appending to a sub-slice can silently overwrite elements in the parent slice if the sub-slice has remaining capacity. When in doubt, use `copy` to get a fully independent slice, or use the three-index form to cap the sub-slice's capacity.

When to Use `copy`

Use `copy` when you need a slice that is fully independent of its source:

- You are returning a sub-slice from a function and the caller should not be able to affect the internal state of the original.
- You are modifying a slice inside a goroutine and need to avoid data races with other goroutines that hold the original.
- You want to snapshot the current contents and continue appending to the original without affecting the snapshot.

Passing Slices to Functions

Passing a slice to a function passes the **slice header** by value: a copy of the pointer, length, and capacity. The function receives its own copy of the header, but both copies point at the same backing array.

This means a function **can** modify the elements in the backing array:

```
func doubleAll(s []int) {
    for i := range s {
        s[i] *= 2
```

```
    }
}

nums := []int{1, 2, 3}
doubleAll(nums)
fmt.Println(nums) // [2 4 6]  --- elements were modified through the shared backing array
```

But a function **cannot** change the caller's slice header (length or capacity) without returning the new slice:

```
func tryAppend(s []int) {
    s = append(s, 99) // modifies the function's local copy of the header
}

nums := []int{1, 2, 3}
tryAppend(nums)
fmt.Println(nums)      // [1 2 3]  --- caller's slice header was not changed
fmt.Println(len(nums)) // 3
```

If a function needs to grow a slice, it must return the new slice and the caller must reassign:

```
func addTrack(playlist []string, track string) []string {
    return append(playlist, track)
}

p := []string{"Sandstorm"}
p = addTrack(p, "Gouryella")
fmt.Println(p) // [Sandstorm Gouryella]
```

Tip: The convention in Go is to accept a slice and return the modified slice, just as `append` does. If your function may grow a slice, return it.

Multidimensional Slices

Go has no built-in 2D slice type. A "matrix" is a slice of slices: `[][]int`. Each row is an independent slice and may have a different length.

```
matrix := [][]int{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
}
fmt.Println(matrix[1][2]) // 6
```

To allocate an m×n matrix dynamically:

```
func newMatrix(rows, cols int) [][]int {
    m := make([][]int, rows)
    for i := range m {
        m[i] = make([]int, cols)
    }
    return m
}
```

Because each row is independent, modifying `m[0]` never affects `m[1]`. There is no guarantee that consecutive rows share a single contiguous backing array — each call to `make` allocates independently. If contiguous

memory matters (for performance or interoperability with C), allocate one large `[]int` and slice it into rows manually.

The `slices` Package (Go 1.21+)

Go 1.21 introduced the `slices` package in the standard library, providing the generic utility functions that previously required hand-rolling or third-party libraries. Import it with `import "slices"`.

Sorting

```
func Sort[S ~[]E, E cmp.Ordered](x S)                    // sort x in ascending order in place
func SortFunc[S ~[]E, E any](x S, cmp func(a, b E) int) // sort x using a custom comparison
```

```
tracks := []string{"The Sound of Silence", "Bad Apple!!", "Sandstorm", "Flaming June"}
slices.Sort(tracks)
fmt.Println(tracks) // [Bad Apple!! Flaming June Sandstorm The Sound of Silence]
```

`Sort` works with any `cmp.Ordered` type: integers, floats, and strings. For custom types or reverse order, use `SortFunc`, which takes a comparison function returning a negative number, zero, or a positive number (the same three-way contract as Java's `Comparator.compare`).

The `cmp` package (Go 1.21) supplies a ready-made three-way comparison so you do not have to hand-roll one (and risk overflow from `a - b`). Import it with `import "cmp"`.

```
func Compare[T cmp.Ordered](x, y T) int // -1 if x<y, 0 if x==y, +1 if x>y
```

```
scores := []int{42, 7, 99, 13}
slices.SortFunc(scores, func(a, b int) int {
    return cmp.Compare(b, a) // cmp.Compare is safe; b-a can overflow
})
fmt.Println(scores) // [99 42 13 7]
```

Searching and Testing

```
func Contains[S ~[]E, E comparable](s S, v E) bool // true if v is in s
func Index[S ~[]E, E comparable](s S, v E) int     // index of first occurrence of v, or -1
```

```
genres := []string{"pop", "hip-hop", "indie", "R&B"}
fmt.Println(slices.Contains(genres, "indie"))  // true
fmt.Println(slices.Contains(genres, "metal"))  // false
fmt.Println(slices.Index(genres, "hip-hop"))   // 1
fmt.Println(slices.Index(genres, "metal"))     // -1
```

Other Useful Functions

```
func Compact[S ~[]E, E comparable](s S) S   // remove consecutive duplicate elements
func Collect[E any](seq iter.Seq[E]) []E    // collect an iterator into a slice (Go 1.23)
```

`Compact` is the equivalent of removing consecutive duplicates (like the Unix `uniq` command). `Collect` pairs with the `iter` package covered in Chapter 14.

Tip: Before Go 1.21, sorting a slice required implementing `sort.Interface` (three methods) or using `sort.Slice` with a comparison closure. `slices.Sort` and `slices.SortFunc` are cleaner and type-safe. Prefer the `slices` package for any new code targeting Go 1.21 or later.

`clear` on a Slice (Go 1.21)

The built-in `clear` function was added in Go 1.21. Applied to a slice, it **zeroes all elements** without changing the length or capacity.

```
func clear[T ~[]E, E any](s T)                  // zero all elements; len and cap unchanged
func clear[T ~map[K]V, K comparable, V any](m T) // delete all entries from the map
```

```
s := []int{1, 2, 3, 4, 5}
clear(s)
fmt.Println(s)      // [0 0 0 0 0]
fmt.Println(len(s)) // 5  --- length unchanged
```

`clear` is useful for reusing a slice buffer without releasing the backing array back to the garbage collector. It is **not** the same as `s = s[:0]`, which also keeps the capacity but does not zero the elements (and for slices of pointers or interfaces, would leave stale references that prevent garbage collection).

Wut: `s = s[:0]` resets the length to zero but leaves the old values in the backing array. For slices of integers or floats that does not matter much, but for slices of pointers, interfaces, or structs containing pointers, the stale values keep referenced objects alive longer than you might expect. `clear(s)` followed by `s = s[:0]` properly zeros the elements before shrinking the length.

Try It

Type this in and run it a few times. It builds a play-count map, looks a key up with the comma-ok idiom, sorts the keys for deterministic output, and uses `copy` to take a snapshot that `append` cannot disturb. Watch how the map iteration would jump around if you printed it raw, while the sorted slice stays put.

```
package main

import (
    "fmt"
    "slices"
)

func main() {
    // Build a play-count map, then report it in sorted order.
    plays := map[string]int{
        "Houdini":            910_000_000,   // Dua Lipa
        "Espresso":           1_300_000_000, // Sabrina Carpenter
        "Birds of a Feather": 1_500_000_000, // Billie Eilish
    }

    plays["Houdini"] += 1_000 // maps mutate in place, no reassignment needed

    // comma-ok: distinguish "absent" from "stored zero".
    if n, ok := plays["Texas Hold 'Em"]; ok {
        fmt.Println("found:", n)
    } else {
        fmt.Println("Texas Hold 'Em: not tracked yet")
    }

    // Collect keys into a slice and sort for deterministic output.
    titles := make([]string, 0, len(plays))
    for title := range plays {
```

```
        titles = append(titles, title)
    }
    slices.Sort(titles)

    for _, title := range titles {
        fmt.Printf("%-20s %d\n", title, plays[title])
    }

    // copy breaks aliasing so appends never touch the original.
    snapshot := make([]string, len(titles))
    copy(snapshot, titles)
    snapshot = append(snapshot, "(new arrival)")
    fmt.Println("snapshot:", snapshot)
    fmt.Println("originals intact:", titles)
}
```

The sorted output is deterministic; the snapshot ends with `(new arrival)` while `titles` is unchanged.

Try these modifications:

- Print `plays` directly with `fmt.Println(plays)` and run it several times to watch the iteration order shift.
- Drop the `copy` and append straight into a re-slice of `titles` to see the aliasing trap bite.
- Swap the key-collecting `for range` loop for a `slices.SortFunc` call that orders titles by play count instead of alphabetically.

Key Points

- A map's type is `map[K]V`; `K` must be comparable (supports `==`).
- Declare a map with a literal or `make`; a nil map panics on write but is safe to read.
- Reading a missing key returns the zero value; use `v, ok := m[k]` to distinguish "absent" from "stored zero."
- Map iteration order is randomized by design; sort keys explicitly for deterministic output.
- `clear(m)` (Go 1.21) removes all entries but leaves the map non-nil and reusable.
- Go has no built-in set; use `map[T]struct{}` (zero-byte values) or `map[T]bool` and rely on the keys.
- Arrays are value types; the length is part of the type (`[3]int` and `[4]int` are different types).
- A slice is a three-field header: pointer to backing array, length, and capacity.
- `[]int` is a slice; `[3]int` is an array — the presence of a number is the difference.
- A nil slice and an empty slice both have length zero; a nil slice can be appended to directly.
- `append` always returns a new slice header; always assign the result back.
- When `append` exceeds capacity it allocates a new backing array — slices that shared the old array are not updated.
- `copy` copies `min(len(dst), len(src))` elements and never grows the destination.
- Slicing expressions share the backing array; use `copy` or the three-index form to avoid accidental aliasing.
- Passing a slice to a function passes the header by value; the function can modify elements but cannot change the caller's length or capacity.
- The `slices` package (Go 1.21) provides `Sort`, `SortFunc`, `Contains`, `Index`, and `Compact`; the iterator-bridging `Collect` was added in Go 1.23.
- `clear` on a slice zeroes all elements without changing the length (Go 1.21).

Exercises

1. **Think about it:** In Java, `HashMap<K,V>` requires keys to implement `hashCode()` and `equals()`, and `ArrayList<E>` stores references to boxed objects on the heap. Go's `map[K]V` requires `K` to be comparable

at the language level, and a `[]E` slice stores values directly in the backing array. What are the trade-offs of Go's approach for each collection type? Give one example of a Java key type you cannot use directly as a Go map key, and explain one scenario where storing values directly in a slice (rather than as heap references) matters for performance.

2. **What does this print?**

```go
package main

import "fmt"

func main() {
    catalog := map[string]int{
        "Saltwater":       1_200_000_000,
        "Out Of The Blue": 980_000_000,
    }
    hits := []string{"Out Of The Blue", "Watermelon Sugar", "Saltwater"}
    for _, title := range hits {
        if plays, ok := catalog[title]; ok {
            fmt.Printf("%s: %d\n", title, plays)
        } else {
            fmt.Printf("%s: not found\n", title)
        }
    }
}
```

3. **Calculation:** Given the following code, trace the value of `len(s)` and `cap(s)` after each line. Does any line cause a new backing array to be allocated?

```go
s := make([]int, 2, 5)
s = append(s, 10)
s = append(s, 20)
s = append(s, 30)
s = append(s, 40)
```

4. **Where is the bug?** The following program tries to build a word-frequency map and then print only the words that appear more than once. It compiles but panics at runtime, even though `"Gamemaster"` appears twice. What is wrong, and how do you fix it?

```go
package main

import "fmt"

func main() {
    words := []string{"Gouryella", "Gamemaster", "Flaming June", "Gamemaster", "Sandstorm"}
    var freq map[string]int
    for _, w := range words {
        freq[w]++
    }
    for word, count := range freq {
        if count > 1 {
            fmt.Println(word, count)
        }
    }
}
```

5. **Write a program:** Write a program that reads a slice of song titles and builds a map from the first letter

(as a `string`) to a slice of titles starting with that letter. Use the input `[]string{"Sandstorm", "Bad Apple!!", "Gouryella", "Better Off Alone", "Flaming June", "Sandstorm"}`. Print each letter and its titles in sorted order (sort both the letters and the titles within each group). Collect the map's keys into a slice with a `for range` loop, then use `slices.Sort` for both the keys and the titles within each group.