Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 18

Generics

Java programmers are no strangers to generics — you have been writing `List<T>` and `Map<K,V>` for years. Go added generics in version 1.18, and while the surface syntax looks familiar, the semantics differ in important ways: Go uses **monomorphization** rather than type erasure, introduces **constraints** as a first-class concept, and ships a constraint system powerful enough to express both simple and nuanced type requirements. This chapter covers type parameters, constraints, the tilde syntax, the standard packages that generics unlocked (`slices`, `maps`, `cmp`, `iter`, `unique`), and — critically — when you should reach for a concrete type instead.

Type Parameters

In Java, a generic method looks like:

```
public static <T, U> List<U> map(List<T> list, Function<T, U> f) { ... }
```

In Go, the type parameters move to a **type parameter list** in square brackets immediately after the function name:

```
func Map[T, U any](s []T, f func(T) U) []U {
    result := make([]U, len(s))
    for i, v := range s {
        result[i] = f(v)  // call f on each element and store the result
    }
    return result
}
```

`[T, U any]` declares two type parameters. `any` is the **constraint** — it means "T and U may be any type whatsoever." At the call site, the compiler usually infers the type arguments from the arguments you pass:

```
titles := []string{"Escape", "$100 Bills", "J'ai pas vingt ans !", "J'en ai marre !"}
lengths := Map(titles, func(s string) int { return len(s) })
fmt.Println(lengths) // [6 10 20 15]
```

You can supply them explicitly when inference fails:

```
lengths := Map[string, int](titles, func(s string) int { return len(s) })
```

Generic Types

Type parameters work on types as well as functions. A generic `Stack` is a classic example:

```
type Stack[T any] struct {
    items []T
```

```
}

func (s *Stack[T]) Push(v T) {
    s.items = append(s.items, v)  // append v to the top of the stack
}

func (s *Stack[T]) Pop() (T, bool) {
    if len(s.items) == 0 {
        var zero T              // zero value of T
        return zero, false
    }
    top := s.items[len(s.items)-1]
    s.items = s.items[:len(s.items)-1]
    return top, true
}
```

Instantiate it with a concrete type:

```
var playlist Stack[string]
playlist.Push("Escape")
playlist.Push("J'ai pas vingt ans !")
v, ok := playlist.Pop()
fmt.Println(v, ok) // J'ai pas vingt ans ! true
```

Tip: Unlike Java, Go generic types are instantiated with the concrete type written at the declaration site: `Stack[string]`, not `new Stack<>()`. There is no diamond operator because Go does not use constructors.

Constraints

A **constraint** is an interface that limits which types may be used as a type argument. Every type parameter has exactly one constraint. You cannot omit the constraint — if you want a type parameter to accept any type, you must write `any` explicitly; there is no implicit "unconstrained."

`any`

`any` is the least restrictive constraint: it permits every type. A type parameter constrained to `any` supports only the operations that every type supports — assignment and passing as a function argument. You cannot call methods on it, compare it with `==`, or use it as a map key.

`comparable`

`comparable` is a built-in constraint that permits any type that supports `==` and `!=`. It is the constraint required for map keys:

```
func Contains[T comparable](s []T, v T) bool {
    for _, elem := range s {
        if elem == v {  // only valid because T is comparable
            return true
        }
    }
    return false
}
```

Trap: `any` and `comparable` are not interchangeable. If you write `func NewCache[K any, V any]() map[K]V`, the compiler rejects it because map keys must be `comparable`. Change `K any` to `K comparable`.

Custom Constraint Interfaces

A constraint can be any interface. The interface may include method requirements, type union elements, or both.

A method constraint requires the type to have a specific method:

```
type Stringer interface {
    String() string  // the type must have a String() string method
}

func PrintAll[T Stringer](items []T) {
    for _, item := range items {
        fmt.Println(item.String())  // safe because T is constrained to Stringer
    }
}
```

A type union constraint lists the exact types permitted:

```
type Number interface {
    int | int32 | int64 | float32 | float64  // any of these five types
}

func Sum[T Number](s []T) T {
    var total T
    for _, v := range s {
        total += v  // + is defined for all types in Number
    }
    return total
}
```

You can combine union elements with method requirements in a single constraint interface, though this is uncommon in practice.

Tip: If you are hunting for Go's equivalent of Java's wildcards, stop — there isn't one. Java has use-site variance (`List<? extends Number>`, `List<? super Integer>`) and bounded type parameters (`<T extends Comparable<T>>`). Go has no wildcards and no variance at all, declaration-site or use-site. The constraint interface is the whole story: it plays the role that both bounded type parameters and wildcards play in Java. Where a Java API might take `List<? extends T>` to stay flexible about the element type, idiomatic Go just writes a generic function with a type parameter and a constraint, e.g. `func Sum[T Number](s []T) T`.

The Tilde Syntax

A type union like `int | float64` is too narrow: it excludes user-defined types whose **underlying type** is `int` or `float64`. Consider:

```
type BPM int
```

`BPM` is not `int` — it is a distinct named type. A function constrained to `int` will not accept `BPM`, even though `BPM` behaves exactly like an integer.

The **tilde** prefix `~T` means "any type whose underlying type is T":

```
type Numeric interface {
    ~int | ~int32 | ~int64 | ~float32 | ~float64
}

func Max[T Numeric](a, b T) T {
    if a > b {
        return a
    }
    return b
}
```

Now `Max` works with `BPM` as well as `int`:

```
type BPM int

a := BPM(128)
b := BPM(140)
fmt.Println(Max(a, b)) // 140
```

Tip: Standard library constraints always use `~T` rather than bare `T` in union elements. `cmp.Ordered` (see below) is defined with `~int | ~int8 | ... | ~string` so that user-defined types work automatically.

Wut: Java generics use **type erasure** — the type parameter is replaced by `Object` (or the bound) at compile time and checked only at the boundaries. At runtime, a `List<String>` and a `List<Integer>` are the same `List`. Go uses **monomorphization** — the compiler generates a distinct instantiation for each unique set of type arguments (or uses a shared representation for pointer-shaped types). The upshot: Go generics have no hidden boxing cost for value types like `int`, and there is no "unchecked cast" at runtime.

`comparable` vs `any` — Map Keys

Map keys must be `comparable`. If you write a generic function that creates or accepts a map, the key type parameter must be constrained to `comparable`.

```
// MapFromSlice builds a map from a slice of keys and a transform function.
func MapFromSlice[K comparable, V any](keys []K, f func(K) V) map[K]V {
    m := make(map[K]V, len(keys))
    for _, k := range keys {
        m[k] = f(k)  // build the map entry
    }
    return m
}

songs := []string{"Escape", "$100 Bills", "J'ai pas vingt ans !", "J'en ai marre !"}
lengths := MapFromSlice(songs, func(s string) int { return len(s) })
fmt.Println(lengths["J'ai pas vingt ans !"]) // 20
```

`any` is deliberately weaker than `comparable`. A `[]int` satisfies `any` but does not satisfy `comparable`. This distinction is enforced at compile time — you cannot accidentally use a non-comparable type as a map key.

The `slices` Package (Go 1.21)

Chapter 7 introduced the `slices` package. Now that you understand type parameters and constraints, the signatures make sense:

```
func Sort[S ~[]E, E cmp.Ordered](x S)                    // sort x in place; E must be ordered
func SortFunc[S ~[]E, E any](x S, cmp func(a, b E) int)  // sort using a custom comparator
func Contains[S ~[]E, E comparable](s S, v E) bool       // true if v appears in s
func Index[S ~[]E, E comparable](s S, v E) int           // first index of v, or -1
func Compact[S ~[]E, E comparable](s S) S                // remove consecutive duplicates
```

`S ~[]E` uses the tilde to accept any slice type whose element is `E`, including user-defined slice types like `type Playlist []string`. `E cmp.Ordered` requires elements that support `<`, `>`, and `==`. `E comparable` requires only `==`.

The iterator-based helpers landed later, in Go 1.23, once the `iter` package existed:

```
func Collect[E any](seq iter.Seq[E]) []E         // collect an iterator into a slice (Go 1.23)
func Sorted[E cmp.Ordered](seq iter.Seq[E]) []E  // collect and sort (Go 1.23)
```

Tip: `slices.Sort` works without a comparator because `cmp.Ordered` guarantees the `<` operator. `slices.SortFunc` works on any element type because it delegates ordering to the comparator you supply. Use `Sort` for simple value types, `SortFunc` for structs.

The `maps` Package (Go 1.21)

The `maps` package complements the maps you met in Chapter 7, just as `slices` does for slices. The full signatures reveal the constraints:

```
func Clone[M ~map[K]V, K comparable, V any](m M) M  // shallow copy of m (Go 1.21)
```

`Map ~map[K]V` uses the tilde to accept any named map type, not just `map[K]V` directly. `K comparable` is required because map keys must be comparable.

The iterator-returning `Keys` and `Values` are newer: the standard library originally exposed slice-returning versions in the experimental `golang.org/x/exp/maps`, but the versions promoted into the standard `maps` package in Go 1.23 return `iter.Seq` so they can be ranged over directly:

```
func Keys[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[K]    // iterator over keys
func Values[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[V]  // iterator over values
```

`Keys` and `Values` return iterators — see the `iter` section below.

The `cmp` Package (Go 1.21)

Chapter 7 introduced `cmp.Compare` and `cmp.Ordered`. The `cmp.Ordered` constraint is defined as:

```
// package cmp
type Ordered interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64 |
        ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~uintptr |
        ~float32 | ~float64 |
        ~string
}

func Compare[T Ordered](x, y T) int // returns -1, 0, or +1
func Less[T Ordered](x, y T) bool   // true if x < y
```

Every element in `Ordered` uses `~` so that user-defined types like `type BPM int` or `type Title string` satisfy the constraint automatically.

The `iter` Package (Go 1.23)

Go 1.23 introduced the `iter` package and **range-over-func** — the ability to `range` over a function value rather than a slice or map.

Iterator Types

```
// package iter
type Seq[V any] func(yield func(V) bool)        // single-value iterator
type Seq2[K, V any] func(yield func(K, V) bool) // two-value iterator (key and value)
```

An iterator is just a function that accepts a **yield callback**. The iterator calls `yield(v)` for each element. If `yield` returns `false`, the iteration should stop — this is how `break` works inside a `range` loop over an iterator.

Writing an Iterator

```
// ArtistTitles yields "Title (Artist)" strings for a slice of tracks.
func ArtistTitles(tracks []Track) iter.Seq[string] {
    return func(yield func(string) bool) {
        for _, t := range tracks {
            if !yield(t.Title + " (" + t.Artist + ")") {  // stop if yield returns false
                return
            }
        }
    }
}
```

Range Over Func

Once you have an `iter.Seq[V]` or `iter.Seq2[K,V]`, you can loop over it with `range`:

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

func main() {
    tracks := []Track{
        {Title: "Escape",              Artist: "Jaroslav Beck"},
        {Title: "$100 Bills",          Artist: "Jaroslav Beck"},
        {Title: "J'ai pas vingt ans !", Artist: "Alizée"},
        {Title: "J'en ai marre !",     Artist: "Alizée"},
    }

    for s := range ArtistTitles(tracks) {
        fmt.Println(s)
    }
    // Escape (Jaroslav Beck)
    // $100 Bills (Jaroslav Beck)
    // J'ai pas vingt ans ! (Alizée)
```

```
    // J'en ai marre ! (Alizée)
}
```

`break` inside the loop causes `yield` to return `false`, which signals the iterator to stop.

`iter.Seq2`

`iter.Seq2[K,V]` is for iterators that yield two values, like index and element. `maps.Keys` and `maps.Values` return `iter.Seq[K]` and `iter.Seq[V]` respectively; `slices.All` (Go 1.23) returns `iter.Seq2[int, E]` yielding index–element pairs.

```
// package slices (Go 1.23)
func All[S ~[]E, E any](s S) iter.Seq2[int, E]  // yields (index, element) pairs
```

```
for i, title := range slices.All([]string{
    "Escape", "J'ai pas vingt ans !", "J'en ai marre !",
}) {
    fmt.Printf("%d: %s\n", i, title)
}
// 0: Escape
// 1: J'ai pas vingt ans !
// 2: J'en ai marre !
```

Tip: The `yield` function is the Go equivalent of Java's `Consumer<T>` in `Iterable.forEach`, but with a crucial difference: returning `false` from `yield` is how early termination (i.e., `break`) is communicated back to the iterator. Always check the return value of `yield` and return immediately when it is `false`.

The `unique` Package (Go 1.23)

The `unique` package provides **value interning** — it canonicalizes equal values so that identical values share the same memory address. This is useful for reducing memory pressure when the same strings (or other comparable values) appear many times.

```
// package unique
func Make[T comparable](v T) Handle[T]  // intern v; equal values return the same Handle

type Handle[T comparable] struct { /* opaque */ }

func (h Handle[T]) Value() T  // retrieve the interned value
```

Two `Handle` values are equal (via `==`) if and only if their underlying values are equal. This means handles can be used as map keys to efficiently deduplicate values:

```
import "unique"

func main() {
    a := unique.Make("Alizée")
    b := unique.Make("Alizée")
    c := unique.Make("Jaroslav Beck")

    fmt.Println(a == b) // true  --- same underlying string
    fmt.Println(a == c) // false --- different strings

    fmt.Println(a.Value()) // Alizée
}
```

Tip: `unique` is particularly useful for reducing allocations when a program repeatedly interns the same short strings (e.g., artist names or tag values from a high-volume stream). Handles are pointer-sized and comparable, so they can serve as efficient map keys in place of full string values.

Generic Type Aliases (Go 1.24)

Go 1.24 added support for **generic type aliases**. Before 1.24, a type alias could not have its own type parameters. Now it can:

```
// A Pair is an alias for a two-element struct.
type Pair[A, B any] = struct {
    First  A
    Second B
}

// A StringMap is an alias for a map with string keys.
type StringMap[V any] = map[string]V
```

Generic type aliases are useful when you want a shorter or domain-specific name for a parameterized type from another package:

```
type Result[T any] = struct {
    Value T
    Err   error
}
```

Wut: A generic type alias uses `=` in the definition, just like a non-generic alias. Without `=` you are defining a new named type, not an alias — the difference matters for method sets and assignability.

When NOT to Use Generics

Generics add expressive power, but they also add complexity. The Go team's guidance, reinforced by experience with the 1.18 release, is to prefer concrete types unless you genuinely need the abstraction.

Use generics when:

- You are writing a utility function that operates uniformly on multiple types with a shared structure (e.g., `Map`, `Filter`, `Reduce` over slices).
- You are building a container type (`Stack[T]`, `Queue[T]`) that does not care what it holds.
- You are writing library code that needs to work across a wide range of user-defined types.

Do not use generics when:

- The function only ever needs one concrete type. Write `func Sum(s []int) int`, not `func Sum[T ~int](s []T) T`.
- An `interface{}` / `any` parameter is simpler and the type is checked at runtime anyway (e.g., encoding libraries).
- The constraint is so complex that callers struggle to satisfy it.
- You are chasing "zero duplication" in application code where two concrete implementations are clearer than one generic one.

Trap: A common mistake when learning Go generics is over-constraining type parameters. If your function only calls `String()` on `T`, constrain `T` to `fmt.Stringer`, not to a union of every concrete type you think callers might use. Narrow constraints keep the function flexible; wide unions tie it to a fixed list.

Tip: Java generics are primarily a **type safety** mechanism — they prevent `ClassCastException` at runtime. Go generics serve the additional role of enabling **static dispatch**: the compiler can inline and optimize generic code in ways that `any` + type assertions cannot. But if you do not need the optimization and the types are few, a simple `interface` with a method set is often more idiomatic Go than a type-parameterized function.

Try It

Type this in and run it. It pulls together a generic `Set[T comparable]`, the iterator-based `maps.Keys`, and `slices.Sorted/slices.SortFunc` with `cmp.Compare` — the generics-powered standard library you met above, working as one program.

```
package main

import (
    "cmp"
    "fmt"
    "maps"
    "slices"
)

// Set is a generic set backed by a map; equal values are stored once.
type Set[T comparable] struct {
    m map[T]struct{}
}

func NewSet[T comparable]() *Set[T] {
    return &Set[T]{m: make(map[T]struct{})}
}

func (s *Set[T]) Add(v T) {
    s.m[v] = struct{}{}  // empty struct is a zero-byte placeholder
}

func (s *Set[T]) Contains(v T) bool {
    _, ok := s.m[v]
    return ok
}

func main() {
    plays := map[string]int{
        "Escape":     12,
        "$100 Bills": 7,
        "Legend":     3,
    }

    // maps.Keys returns an iter.Seq[string]; slices.Sorted collects and sorts it.
```

```
    titles := slices.Sorted(maps.Keys(plays))
    fmt.Println("titles:", titles)

    set := NewSet[string]()
    for _, t := range titles {
        set.Add(t)
    }
    set.Add("Escape")  // duplicate, silently ignored

    fmt.Println("has Escape:", set.Contains("Escape"))

    // slices.SortFunc with cmp.Compare orders entries by play count, descending.
    type entry struct {
        title string
        plays int
    }
    entries := make([]entry, 0, len(plays))
    for t, n := range plays {
        entries = append(entries, entry{t, n})
    }
    slices.SortFunc(entries, func(a, b entry) int {
        return cmp.Compare(b.plays, a.plays)
    })
    fmt.Println("most played:", entries[0].title)
}
```

It prints `titles: [$100 Bills Escape Legend]`, `has Escape: true`, and `most played: Escape`.

Try these modifications:

- Add a `Values() []T` method to `Set[T]` and a `Len() int` method, then print the set's size.
- Add a second `comparable` type parameter and make `Set` hold structs instead of strings.
- Swap `cmp.Compare(b.plays, a.plays)` for `cmp.Compare(a.plays, b.plays)` and see the order flip to ascending.

Key Points

- Type parameters are declared in square brackets after the function or type name: `func Foo[T any](...) ...`.
- Every type parameter requires a constraint; `any` permits all types, `comparable` permits types that support `==`.
- Constraint interfaces may contain method requirements, type union elements (`int | string`), or both.
- The tilde prefix `~T` means "any type whose underlying type is T" — essential for user-defined types like `type BPM int`.
- Map key type parameters must be constrained to `comparable`, not `any`.
- Go uses **monomorphization** (concrete code per instantiation), not type erasure; value types like `int` are never boxed.
- The `slices`, `maps`, and `cmp` packages arrived in Go 1.21 as the standard library's first-class use of generics; their `~T` constraints accept user-defined slice and map types. The iterator-based helpers (`slices.Collect`, `maps.Keys`, `maps.Values`) came later, in Go 1.23, once the `iter` package existed.
- `iter.Seq[V]` and `iter.Seq2[K,V]` (Go 1.23) are the iterator types; write an iterator by returning a function that calls a `yield` callback and checks its return value.
- `unique.Make[T]` (Go 1.23) interns comparable values; equal values share one canonical `Handle`.
- Generic type aliases (`type Alias[T any] = OtherType[T]`) are supported from Go 1.24.

- Prefer concrete types and interfaces over generics in application code; reach for generics when writing containers or utilities that must work uniformly across many types.

Exercises

1. **Think about it:** Java generics use **type erasure**: at runtime, `List<String>` and `List<Integer>` are both just `List`. Generic type information is only available at compile time. Go generics use **monomorphization** (or a shared pointer-shaped representation): the compiler may generate distinct code for each instantiation. Describe one concrete advantage and one concrete disadvantage of each approach. How does type erasure affect what you can do with a Java generic type at runtime (e.g., `instanceof List<String>`)? Does Go have the same limitation?

2. **What does this print?**

```go
package main

import "fmt"

func Filter[T any](s []T, keep func(T) bool) []T {
    var out []T
    for _, v := range s {
        if keep(v) {
            out = append(out, v)
        }
    }
    return out
}

type BPM int

func main() {
    beats := []BPM{72, 128, 96, 140, 80}
    fast := Filter(beats, func(b BPM) bool { return b >= 120 })
    fmt.Println(fast)

    words := []string{"Escape", "J'ai pas vingt ans !", "J'en ai marre !", "$100 Bills"}
    long := Filter(words, func(s string) bool { return len(s) > 7 })
    fmt.Println(long)
}
```

3. **Calculation:** A function with the signature `func Reduce[T, U any](s []T, init U, f func(U, T) U) U` folds a slice into a single value. Trace the execution of `Reduce([]int{1, 2, 3, 4}, 0, func(acc, v int) int { return acc + v })`. What is the concrete type bound to `T`? What is the concrete type bound to `U`? What value does the function return, and what are the intermediate values of `acc` after each call to `f`?

4. **Where is the bug?**

```go
package main

import "fmt"

type Playlist []string

func Dedupe[T any](s []T) []T {
    seen := make(map[T]bool)
```

```
    var out []T
    for _, v := range s {
        if !seen[v] {
            seen[v] = true
            out = append(out, v)
        }
    }
    return out
}

func main() {
    p := Playlist{
        "Escape", "J'ai pas vingt ans !", "Escape",
        "J'en ai marre !", "J'ai pas vingt ans !",
    }
    fmt.Println(Dedupe(p))
}
```

5. **Write a program:** Implement a generic `Set[T comparable]` type backed by a `map[T]struct{}`. It should support three methods: `Add(v T)` (add an element), `Contains(v T) bool` (membership test), and `Values() []T` (return all elements as a slice in any order). In `main`, create a `Set[string]`, add the four song titles “Escape”, “$100 Bills”, “J’ai pas vingt ans !”, and “J’en ai marre !”, add “J’ai pas vingt ans !” a second time, and print the length of the set and whether it contains “Escape” and “Legend”.