# Generics in Go

## Session 18

Golang course by Exadel

15 Dec 2022

Sergio Kovtunenko
Lead backend developer, Exadel

# Agenda

- ▶️ Quickly: what are generics?
- ▶️ Typical use-cases for the generic code
- ▶️ How to understand generics specification?
- ▶️ How do generics in Go compare to generics in other languages?
- ▶️ Future prediction

2

# Quickly: what are generics?

▶️ Generic programming enables the representation of functions and data structures in a generic form, **with types factored out**.

▶️ **Interface types** in Go are a form of generic programming. (*That is how the standard library's `sort.Sort` function works.*)

▶️ Go has two general purpose generic data structures built into the language: `slices` and `maps` (+ chan).

▶️ `Slices` and `maps` can hold values of any data type, with static type checking for values stored and retrieved. The values are stored as themselves, not as interface types.

Source: "Why Generics?" by Ian Lance Taylor (31 July 2019) (https://go.dev/blog/why-generics)

# Typical use-cases for the generic code (pt. 1)

## 👍 When using language-defined **container** types:

```
func MapKeys[Key comparable, Val any](m map[Key]Val) []Key {
    s := make([]Key, 0, len(m))
    for k := range m {
        s = append(s, k)
    }
    return s
}
```

## 👍 General purpose **data structures**:

```
type Tree[T any] struct {
    cmp  func(T, T) int
    root *node[T]
}

type node[T any] struct {
    left, right  *node[T]
    val          T
}
func (bt *Tree[T]) Insert(val T) bool { /* ........*/ }
```

Source: "When To Use Generics" by Ian Lance Taylor (12 April 2022) (https://go.dev/blog/when-generics) 4

# Typical use-cases for the generic code (pt. 2)

👍 There are many other functions that we could write generically, such as:

- Find smallest/largest element in slice
- Compute union/intersection of maps
- Find shortest path in node/edge graph
- Apply transformation function to slice/map, returning new slice/map

👍 There are also examples that are specific to Go with its strong support for concurrency:

- Combine two channels into a single channel
- Call a list of functions in parallel, returning a slice of results

👍 For type parameters, prefer `functions` to `methods`. Because `methods` can't be extra parametrized.

Source: "Why Generics?" by Ian Lance Taylor (31 July 2019) (https://go.dev/blog/why-generics)

# How are generics implemented in Go?

- Virtual method table
- Monomorphization

▶️ The compiler can choose whether to compile each instantiation separately or whether to compile reasonably similar instantiations as a single implementation.

▶️ The single implementation approach is similar to a function with an interface parameter.

▶️ **What can be parametrized?**

- *Types* (including.... interfaces!)
- *Functions* (not methods)
    - Go permits a generic type to have **methods**, but, **other than the receiver**, the arguments to those methods **cannot use parameterized types**.

Source: "Frequently Asked Questions (FAQ)" (https://go.dev/doc/faq)

# Interfaces: basic and general

▶️ Interfaces can be:

- **Basic** interfaces
- **General** interfaces
  - Interfaces that are not basic may only be used as type constraints, or as elements of other interfaces used as constraints.
  - They cannot be the types of values or variables, or components of other, non-interface types:

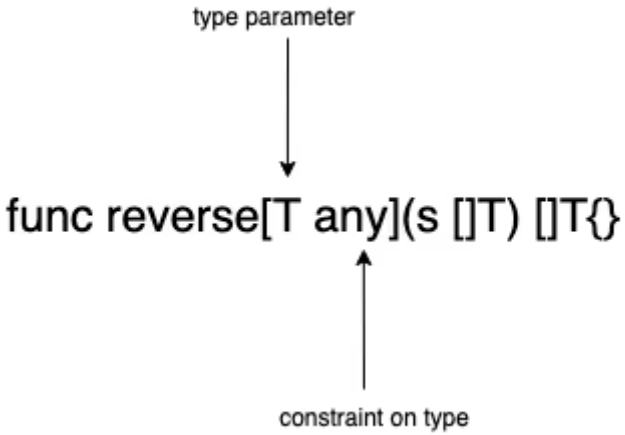
```
type Float interface {
    ~float32 | ~float64
}
var x Float                     // illegal: Float is not a basic interface
var x interface{} = Float(nil)  // illegal
type Floatish struct {
    f Float                     // illegal
}
```

Source: "The Go Programming Language Specification" (Version of June 29, 2022)

(https://go.dev/ref/spec)

# General interfaces

- General **interfaces** can be perceived as "*type sets*" - they form **Type Constraints**
  - Good example: `golang.org/x/exp/constraints`
- Look at the visualization:




Source: "Understanding generics in Go 1.18" (https://blog.logrocket.com/understanding-generics-go-1-18/)

8

# Type parameters and arguments

- Type **parameter**: `T` in the example definition `[T comparable]`
  - Type **argument**
- Type **constraint**: `comparable` in the example definition `[T comparable]`
  - Type constraints are always Go interfaces!

```
type T[P *C] ... // Problem!
// Solution:
type T[P interface{*C}] ... // fix: wrap into interface
type T[P *C,] ... // fix: check the comma at the end!
```

- Type **instantiation**:
  - A generic function or type is **instantiated** by *substituting type arguments* for the type parameters
- Type **approximation**: `~int | ~string`

Source: "The Go Programming Language Specification" (Version of June 29, 2022)

(https://go.dev/ref/spec)

# Omitting interface{ ... } in type constraints

▶️ In the **type constraints**:

- If the **constraint** is an interface literal of the form `interface{E}` where `E` is an embedded type element (not a method), in a type parameter list the enclosing `interface{ ... }` may be omitted for convenience:

```
[T []P]                    // equals to: [T interface{[]P}]
[T ~int]                   // equals to: [T interface{~int}]
[T int|string]             // equals to: [T interface{int|string}]
type Constraint ~int       // illegal: ~int is not inside a type parameter list
```

# Predefined interface type `comparable`

▶️ The predeclared interface type comparable denotes the set of all non-interface types that are comparable. Specifically, a type T implements comparable if:

- T is not an interface type and T supports the operations == and !=; or
- T is an interface type and each type in T's type set implements comparable.

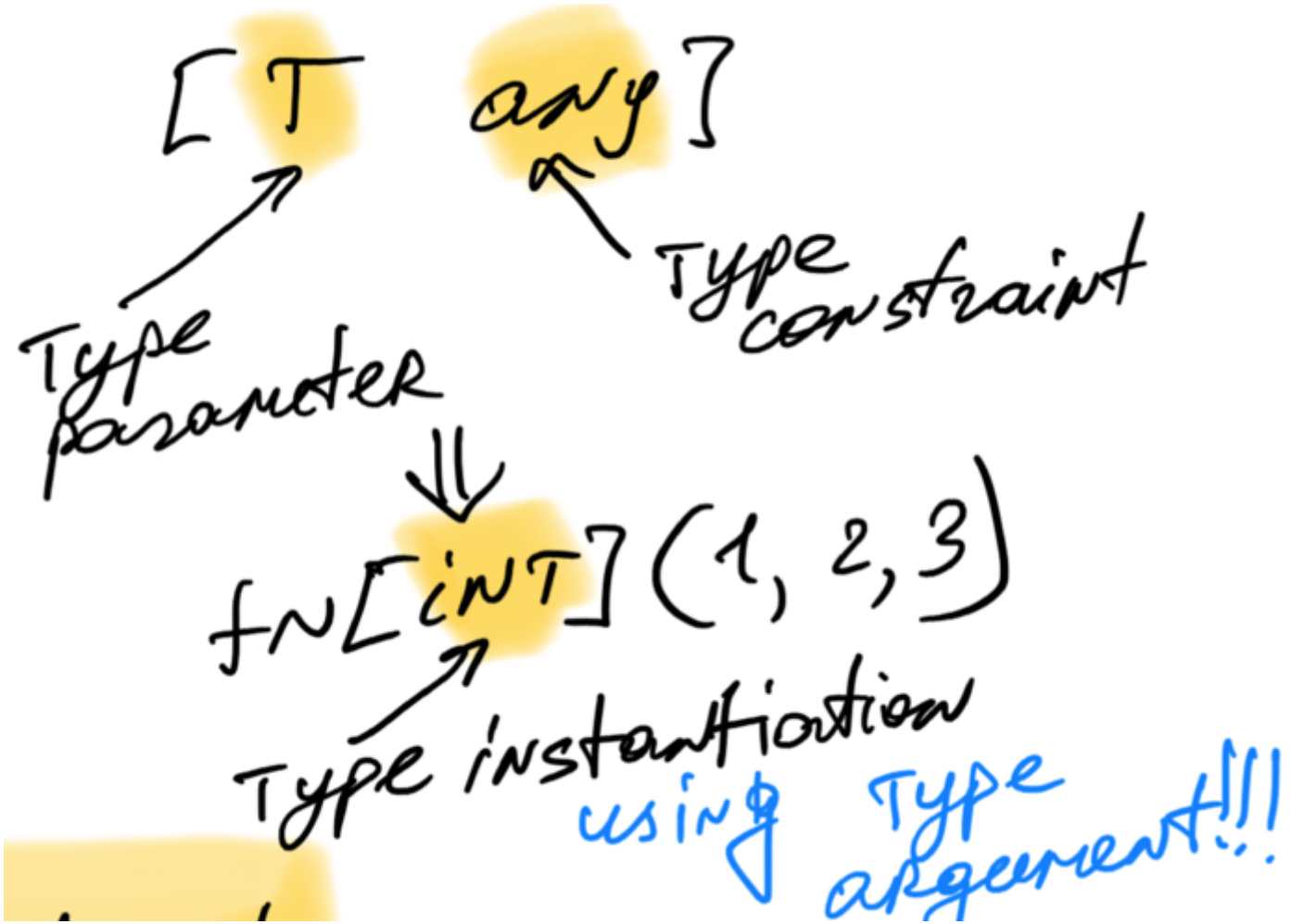
▶️ Example:

```
int                           // implements comparable
[]byte                        // does not implement comparable (slices cannot be compared)
interface{}                   // does not implement comparable (see above)
interface{ ~int | ~string }   // type parameter only: implements comparable
interface{ comparable }       // type parameter only: implements comparable
interface{ ~int | ~[]byte }   // type parameter only: does not implement comparable
                              // (in the example, not all types in the type set are comparable)
```

11

# How to understand generics specification? (pt. 4)

▶️ Recap:




12

# Special cases

▶️ Type constraint: any

- `type any = interface{}`

▶️ Type constraint: `comparable`

- `type comparable interface{ comparable }`

13

# Tricky cases

▶️ Type switches

- Check the code

▶️ Implementing Generic Interfaces

- Types don't actually implement generic interfaces, they implement instantiations of generic interfaces.
    - You can't use a generic type (including interfaces) without instantiation.
- more details: https://stackoverflow.com/a/72050933 (https://stackoverflow.com/a/72050933)

▶️ Hack to simulate Generic methods: `Facilitators` pattern

- More details: https://rakyll.org/generics-facilititators/ (https://rakyll.org/generics-facilititators/)

# Future prediction

▶️ More libs will be migrated and will start using generics in 2022-2023

▶️ Here are some proposals for generic packages, functions, and data structures:

- **constraints**, providing type constraints (#47319)
- **maps**, providing generic map functions (#47330)
- **slices**, providing generic slice functions (#47203)
- **sort.SliceOf**, a generic sort implementation (#47619)
- **sync.PoolOf** and other generic concurrent data structures (#47657)

# Next session...

▶️ Session19:

## Possible ways to design flexible APIs in Go

- “Accept interface, return concrete type” rule
- Breaking API changes
- Rigid API
- Config struct
- Functional options
- Fluent API

16

# Thank you

Golang course by Exadel

15 Dec 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)