Let's Go generic

Go 1.18 type parameters

Agenda

- Introduction
- Generic programming substitutes (before Go 1.18)
- History of Go generics
- Dive into Go 1.18 type parameters
- Use case: type-safe set implementation
- Generics usage recommendations
- Future of Go generics

Introduction

Introduction

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters [36]
- Go 1.18 brings generic programming mechanism - type parameters

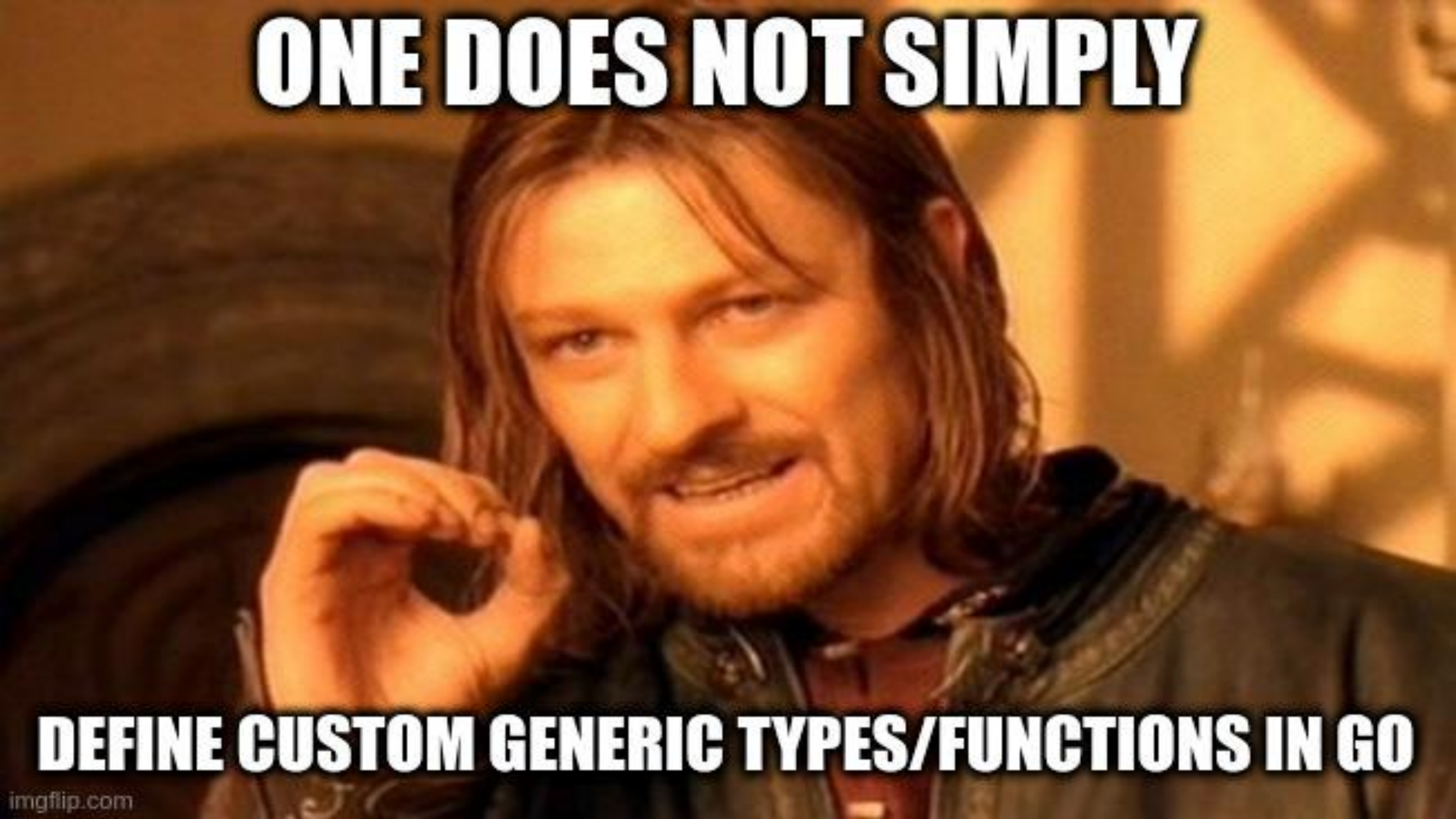
Before Go 1.18

Generic programming substitutes

codilime

Generic programming substitutes

- Go has generic constructs:
    - Types: *slice, map, channel*
    - Functions: *append(), copy(), delete(), len(), cap(), make(), new(), complex(), real(), imag(), close(), print(), println()*
- What about custom generic types/functions?


ONE DOES NOT SIMPLY
DEFINE CUSTOM GENERIC TYPES/FUNCTIONS IN GO
imgflip.com

Generic programming substitutes

- Approaches we had so far:
  - Manual code duplication
  - Code duplication via code generation
  - Operating on empty interfaces (*interface{}*) and using type assertions
  - Operating on empty interfaces (*interface{}*) and using reflections
  - Operating on defined interfaces
- Each approach has its own disadvantages

Generic programming substitutes - examples

- Task: implement function returning maximum number in given slice

Manual code duplication

```go
func MaxInt(s []int) int {
    if len(s) == 0 {
        return 0
    }

    max := s[0]
    for _, v := range s[1:] {
        if v > max {
            max = v
        }
    }
    return max
}

func ExampleMaxInt() {
    m := MaxInt([]int{4, -8, 15})
    fmt.Println(m) // 15
}
```

```go
func MaxFloat64(s []float64) float64 {
    if len(s) == 0 {
        return 0
    }

    max := s[0]
    for _, v := range s[1:] {
        if v > max {
            max = v
        }
    }
    return max
}

func ExampleMaxFloat64() {
    m := MaxFloat64([]float64{4.1, -8.1, 15.1})
    fmt.Println(m) // 15.1
}
```

Manual code duplication

- Key disadvantages:
  - Lots of manual labor
  - Code duplication lowers maintainability

Code generation

- We can automate such code duplication via code generation
- There are even tools for that, e.g. https://github.com/cheekybits/genny
- Key disadvantages:
  - It complicates project build
  - It increases compilation times

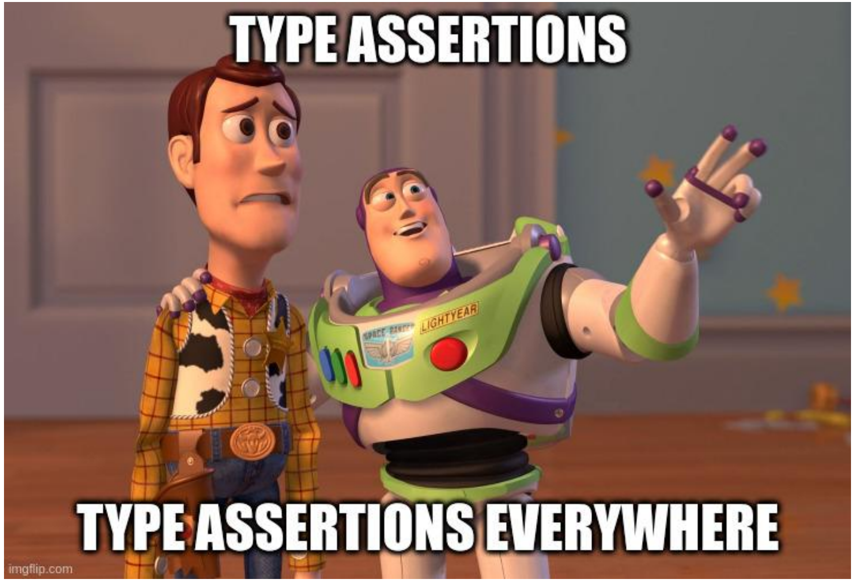
Empty interfaces and type assertions (1/2)

```
func MaxNumber(s []interface{}) (interface{}, error) {
   if len(s) == 0 {
      return nil, errors.New("no values given")
   }
  switch first := s[0].(type) {
   case int:
      max := first
      for _, rawV := range s[1:] {
         v := rawV.(int)
         if v > max {
            max = v
         }
      }
      return max, nil
   [...]
```

```
   [...]
   case float64:
      max := first
      for _, rawV := range s[1:] {
         v := rawV.(float64)
         if v > max {
            max = v
         }
      }
     return max, nil
   default:
      return nil, fmt.Errorf("unsupported element type of given
slice: %T", first)
   }
}
```

Empty interfaces and type assertions (2/2)

```
func ExampleMaxNumber() {
   m1, err1 := MaxNumber([]interface{}{4, -8, 15})
   m2, err2 := MaxNumber([]interface{}{4.1, -8.1, 15.1})
   fmt.Println(err1, err2)                    // <nil> <nil>
   fmt.Println(m1, m2)                        // 15 15.1
}
```



Key disadvantages:

- Losing type-safety
- Type assertions both in caller and algorithm code
- Caller needs to wrap arguments in *interface{}*

Empty interfaces and reflections (1/2)

```
func MaxNumber(s []interface{}) (interface{}, error) {
   if len(s) == 0 {
      return nil, errors.New("no values given")
   }

   first := reflect.ValueOf(s[0])
   if first.CanInt() {
      max := first.Int()
      for _, ifV := range s[1:] {
         v := reflect.ValueOf(ifV)
         if v.CanInt() {
            intV := v.Int()
            if intV > max {
               max = intV
            }
         }
      }
      return max, nil
   }

   [...]
```

```
   [...]

   if first.CanFloat() {
      max := first.Float()
      for _, ifV := range s[1:] {
         v := reflect.ValueOf(ifV)
         if v.CanFloat() {
            intV := v.Float()
            if intV > max {
               max = intV
            }
         }
      }
      return max, nil
   }

   return nil, fmt.Errorf("unsupported element type of given slice:
%T", s[0])
}
```

Empty interfaces and reflections (2/2)

```
func ExampleMaxNumber() {
    m1, err1 := MaxNumber([]interface{}{4, -8, 15})
    m2, err2 := MaxNumber([]interface{}{4.1, -8.1, 15.1})
    fmt.Println(err1, err2) // <nil> <nil>
    fmt.Println(m1, m2)    // 15 15.1
}
```

Key disadvantages:

- Abysmal readability
- Losing type-safety
- Lower performance than in other approaches

Operating on defined interfaces - max

```go
type ComparableSlice interface {
   // Len is the number of elements in the collection.
   Len() int
   // Less reports whether the element with index i has lower value than
the element with index j.
   Less(i, j int) bool
   // Elem returns the element with index i.
   Elem(i int) interface{}
}

func MaxNumber(s ComparableSlice) (interface{}, error) {
   if s.Len() == 0 {
      return nil, errors.New("no values given")
   }

   max := s.Elem(0)
   for i := 1; i < s.Len(); i++ {
      if s.Less(i-1, i) {
         max = s.Elem(i)
      }
   }

   return max, nil
}
```

```go
type ComparableIntSlice []int

func (s ComparableIntSlice) Len() int { return len(s) }
func (s ComparableIntSlice) Less(i, j int) bool { return s[i] < s[j] }
func (s ComparableIntSlice) Elem(i int) interface{} { return s[i] }

type ComparableFloat64Slice []float64

func (s ComparableFloat64Slice) Len() int { return len(s) }
func (s ComparableFloat64Slice) Less(i, j int) bool { return s[i] < s[j] }
func (s ComparableFloat64Slice) Elem(i int) interface{} {return s[i]}

func ExampleMaxNumber() {
   m1, err1 := MaxNumber(ComparableIntSlice([]int{4, -8, 15}))
   m2, err2 := MaxNumber(
      ComparableFloat64Slice([]float64{4.1, -8.1, 15.1})
   )
   fmt.Println(err1, err2) // <nil> <nil>
   fmt.Println(m1, m2)    // 15 15.1
}
```

Operating on defined interfaces - sort (1/2)

```
func Sort(data sort.Interface) {}

type Interface interface {
    // Len is the number of elements in the collection.
    Len() int

    // Less reports whether the element with index i
    // must sort before the element with index j.
    Less(i, j int) bool

    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

Operating on defined interfaces - sort (2/2)

```go
type IntSlice []int

func (s IntSlice) Len() int {
    return len(s)
}

func (s IntSlice) Less(i, j int) bool {
    return s[i] < s[j]
}

func (s IntSlice) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
```

```go
func ExampleIntSlice() {
    s := []int{4, -8, 15}
    sort.Sort(IntSlice(s))
    fmt.Println(s)          // [-8 4 15]
}
```

Key disadvantages:

- Hard to use: requires defining custom types implementing specific methods

History of Go generics

History of Go generics

"Generics are convenient, but they come at a cost in complexity in the type system and run-time. We haven't yet found a design that gives value proportionate to the complexity, although we continue to think about it." - Go FAQ [1]

History of Go generics

- 2007-09 - Go language idea
- 2009-11 - Go became a public open source project
- 2010-06 - Type functions proposal
- 2011-03 - Generalized types proposal
- 2012-03 - Go 1.0 release
- 2013-10 - Generalized types proposal II
- 2013-12 - Type parameters proposal
- 2016-09 - Compile-time Functions and First Class Types proposal
- 2018-08 - Go 2 Draft Designs containing generics with contracts
- 2020-06 - Type parameters draft design
- 2021-03 - Type parameters proposal
- **2022-03 - Go 1.18 release**

Dive into Go 1.18 type parameters

Type parameters fundamentals [21, 31]

- Type parameter has a type constraint (meta-type)
- Type constraint specifies the permissible type arguments that calling code can use for the respective type parameter
- At compile time the type parameter stands for a single type – the type provided as a type argument by the calling code
- A type argument is valid if it implements type parameter's constraint
- Compiler might infer type argument based on function parameter passed by caller

```go
func FirstElem[T any](s []T) T {
    return s[0]
}

func ExampleFirstElem() {
    s1 := []string{"Go", "rocks"}
    s2 := []int{4, 8, 15}
    s3 := []string{"it", "hoge"}

    r1 := FirstElem[string](s1)
    r2 := FirstElem[int](s2)
    r3 := FirstElem(s3)

    fmt.Println(r1, r2, r3) // Go 4 it
    // Output: Go 4 it
}
```

Type constraints fundamentals [31, 32]

- Type constraint defines set of types
- Type constraint is declared as an interface containing union of types or methods
- The constraint allows any type satisfying the interface
- If all types in the constraint support an operation, that operation may be used with the respective type parameter
- *~T* means the set of all types with underlying type *T*

```go
type StringableFloat interface {
    ~float32 | ~float64 // union of types
    String() string
}

// satisfies StringableFloat type constraints
type MyFloat float64

func (m MyFloat) String() string {
    return fmt.Sprintf("%e", m)
}

func StringifyFloat[T StringableFloat](f T) string {
    return f.String()
}

func ExampleMyFloat() {
    var f MyFloat = 48151623.42
    s := StringifyFloat[MyFloat](f)
    fmt.Println(s) // 4.815162e+07
}
```

codilime

Information Classification: Public

Generic type constraints

- Type constraint can reference other type parameters

```go
type SliceConstraint[E any] interface {
    ~[]E
}

func FirstElem1[S SliceConstraint[E], E any](s S) E
{
    return s[0]
}

func ExampleSlice() {
    s := []string{"Go", "rocks"}
    r1 := FirstElem1(s)
    fmt.Println(r1) // Go
}
```

Inlined type constraints

- Type constraints can be inlined

```go
type SliceConstraint[E any] interface {
    ~[]E
}

func FirstElem1[S SliceConstraint[E], E any](s S) E
{
    return s[0]
}

func FirstElem2[S interface{ ~[]E }, E any](s S) E {
    return s[0]
}

func FirstElem3[S ~[]E, E any](s S) E {
    return s[0]
}

func ExampleSlice() {
    s := []string{"Go", "rocks"}
    r1 := FirstElem1(s)
    r2 := FirstElem2(s)
    r3 := FirstElem3(s)
    fmt.Println(r1, r2, r3) // Go Go Go
}
```

Built-in type constraints

- Built-in type constraints:
  - any - alias for *interface{}*
  - comparable - any type whose values may be used as an operand of the comparison operators == and !=
- Constraints defined in golang.org/x/exp/constraints:
  - Signed - *~int | ~int8 | ~int16 | ~int32 | ~int64*
  - Unsigned - *~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~uintptr*
  - Integer - *Signed | Unsigned*
  - *Float* - *~float32 | ~float64*
  - Complex - *~complex64 | ~complex128*
  - Ordered - *Integer | Float | ~string* (any type that supports the operators < <= >= >)

Max function example

```go
func Max[T constraints.Ordered](s []T) T {
	if len(s) == 0 {
		return *new(T)
	}

	max := s[0]
	for _, v := range s[1:] {
		if v > max {
			max = v
		}
	}
	return max
}
```

```go
func ExampleMax() {
	m1 := Max[int]([]int{4, -8, 15})
	m2 := Max([]float64{4.1, -8.1, 15.1})

	type customInt int
	m3 := Max([]customInt{4, -8, 15})

	fmt.Println(m1, m2, m3) // 15 15.1 15
}
```

Compilation

```
func Max[T constraints.Ordered](s []T) T {}
m := Max([]int{4, -8, 15})
```

Function compilation steps for generic code [9, 31, 32]:

1. Type inference (new)
   - Argument type inference: deduce unknown type arguments from the types of the ordinary arguments
   - Constraint type inference: deduce unknown type arguments from known type arguments
2. Instantiation (new)
   - Replace type parameters with type arguments in entire signature
   - Verify that each type argument satisfies its constraint
   - Instantiate internal function with given type arguments
3. Invocation (as pre Go 1.18)
   - Verify that each ordinary argument can be assigned to its parameter

Use case: type-safe set implementation

Use case: type-safe set implementation

- Task: implement set data structure

Use case: type-safe set implementation (1/3)

```go
// Set implements generic set data structure backed by a hash table.
// It is not thread safe.
type Set[T comparable] struct {
    values map[T]struct{}
}

func NewSet[T comparable](values ...T) *Set[T] {
    m := make(map[T]struct{}, len(values))
    for _, v := range values {
        m[v] = struct{}{}
    }
    return &Set[T]{
        values: m,
    }
}
```

```go
func (s *Set[T]) Add(values ...T) {
    for _, v := range values {
        s.values[v] = struct{}{}
    }
}

func (s *Set[T]) Remove(values ...T) {
    for _, v := range values {
        delete(s.values, v)
    }
}

func (s *Set[T]) Contains(values ...T) bool {
    for _, v := range values {
        _, ok := s.values[v]
        if !ok {
            return false
        }
    }
    return true
}
```

Use case: type-safe set implementation (2/3)

```go
func (s *Set[T]) Union(other *Set[T]) *Set[T] {
    result := NewSet[T](s.Values()...)
    for _, v := range other.Values() {
        if !result.Contains(v) {
            result.Add(v)
        }
    }
    return result
}

func (s *Set[T]) Intersect(other *Set[T]) *Set[T] {
    // pass smaller set first for optimization
    if s.Size() < other.Size() {
        return intersect(s, other)
    }
    return intersect(other, s)
}
```

```go
// intersect returns intersection of given sets. It iterates over
smaller set for optimization.
func intersect[T comparable](smaller, bigger *Set[T]) *Set[T] {
    result := NewSet[T]()
    for k, _ := range smaller.values {
        if bigger.Contains(k) {
            result.Add(k)
        }
    }
    return result
}

func (s *Set[T]) Values() []T {
    return s.toSlice()
}

func (s *Set[T]) toSlice() []T {
    result := make([]T, 0, len(s.values))
    for k := range s.values {
        result = append(result, k)
    }
    return result
```

codilime

Use case: type-safe set implementation (3/3)

```go
func (s *Set[T]) Size() int {
    return len(s.values)
}

func (s *Set[T]) Clear() {
    s.values = map[T]struct{}{}
}

func (s *Set[T]) String() string {
    return fmt.Sprint(s.toSlice())
}
```

```go
func ExampleSet() {
    s1 := NewSet(4, 4, -8, 15)
    s2 := NewSet("foo", "foo", "bar", "baz")
    fmt.Println(s1.Size(), s2.Size())       // 3, 3

    s1.Add(-16)
    s2.Add("hoge")
    fmt.Println(s1.Size(), s2.Size())               // 4, 4
    fmt.Println(s1.Contains(-16), s2.Contains("hoge")) // true, true

    s1.Remove(15)
    s2.Remove("baz")
    fmt.Println(s1.Size(), s2.Size()) // 3, 3
    fmt.Println(len(s1.Values()), len(s2.Values())) // 3, 3

    s3 := NewSet("hoge", "dragon", "fly")
    fmt.Println(s2.Union(s3).Size())    // 5
    fmt.Println(s2.Intersect(s3))       // [hoge]

    s1.Clear()
    s2.Clear()
    fmt.Println(s1.Size(), s2.Size()) // 0, 0
}
```

codilime

Generics usage recommendations

Use cases for generics [30, 32]

- Functions operating on slices, maps, channels of any element type
  - Functions doing calculations on elements of slice or map, e.g. max/min/average/mode/standard deviation
  - Transformation functions for slices or maps, e.g. scale a slice
  - Functions operating on channels, e.g. combine two channels into single channel
- General purpose data structures, e.g. set, multimap, concurrent hash map, graph, tree, linked list
- Functions operating on functions, e.g. call given functions in parallel and return a slice of results
- When the implementation of a common method looks the same for each type

When not to use generics [32]

- When just calling a method on given object - use specific interfaces
  - E.g. func **ReadAll(r io.Reader)** ([]byte, error)
  - Not func **ReadAll[T io.Reader](r T)** ([]byte, error)
- When the implementation of a common method is different for each type
  - E.g. file.Read() and buffer.Read()
- When the operation is different for each type (and requiring specific interfaces is not preferred) - use reflections
  - E.g. func **Marshal(v interface{})** ([]byte, error)
  - Not func **Marshal[T Marshaler](v T)** ([]byte, error)

Usage rules of thumb [32]

- Write a code as usual and refactor it to use type parameters only if you see repeating boilerplate
- When operating on type parameters, prefer functions to methods. For the caller it is easier to pass function than modify types to implement methods.

Future of Go generics

Future of Go generics

- Growing number of libraries using generics:
    - bradenaw/juniper
    - samber/lo
    - mikhailswift/go-collections
    - BooleanCat/go-functional
    - xakep666/unusual_generics
- Go 1.19 (~08.2022): stdlib packages to use generics:
    - golang/exp/constraints
    - golang/exp/maps
    - golang/exp/slices

Summary

- Arguably the biggest shortcoming of Go is going to be resolved soon
- Type parameters is powerful feature, but can be overused
- There was never better time to give Go a go

References (1/3)

- [1] Proposal: Go should have generics (2011-01)
- [2] Type Functions Proposal (2010-06)
- [3] Proposal: Generalized Types (2011-03)
- [4] Proposal: Generalized Types II (2013-10)
- [5] Proposal: Type Parameters (2013-12)
- [6] Proposal: Compile-time Functions and First Class Types (2016-09)
- [7] Generics — Problem Overview (2018-08)
- [8] Draft Design: Contracts 2018-08
- [9] Proposal: Type Parameters (2021-03)
- [10] Proposal: Generic parameterization of array sizes (2021-03)

References (2/3)

- [11] Proposal: Additions to go/ast and go/token to support parameterized functions and types (2021-08)
- [12] Proposal: Additions to go/types to support type parameters (2021-08)
- [13] Summary of Go Generics Discussions
- [14] spec: add generic programming using type parameters #43651 (Jan 12, 2021)
- [15] spec: generics: use type sets to remove type keyword in constraints #45346 (Apr 2021)
- [20] Go FAQ: Why does Go not have generic types?
- [21] Tutorial: Getting started with generics
- [22] Go 2 Generics Feedback
- [23] Go2 status
- [24] Go 1.18 Release Notes
- [25] how to update APIs for generics #48287
- [26] Expectations for generics in Go 1.18

References (3/3)

- [30] GopherCon 2019: Ian Lance Taylor -Generics in Go
- [31] GopherCon 2020: Robert Griesemer - Typing Generic Go
- [32] GopherCon 2021: Robert Griesemer & Ian Lance Taylor - Generics!
- [33] Golang Poland #4 - Bill Kennedy - Generics Draft Proposal Review
- [34] Go Day 2021 on Google Open Source Live | Using Generics in Go
- [35] github.com/ardanlabs/gotraining - generics
- [36] https://en.wikipedia.org/wiki/Generic_programming

Thank you

codilime

Daniel Furman
Software Engineer

Gophers Slack: @danielfurman
GitHub: @danielfurman
Twitter: @danielfurman88
LinkedIn: @danielfurman8

daniel.furman@codilime.com