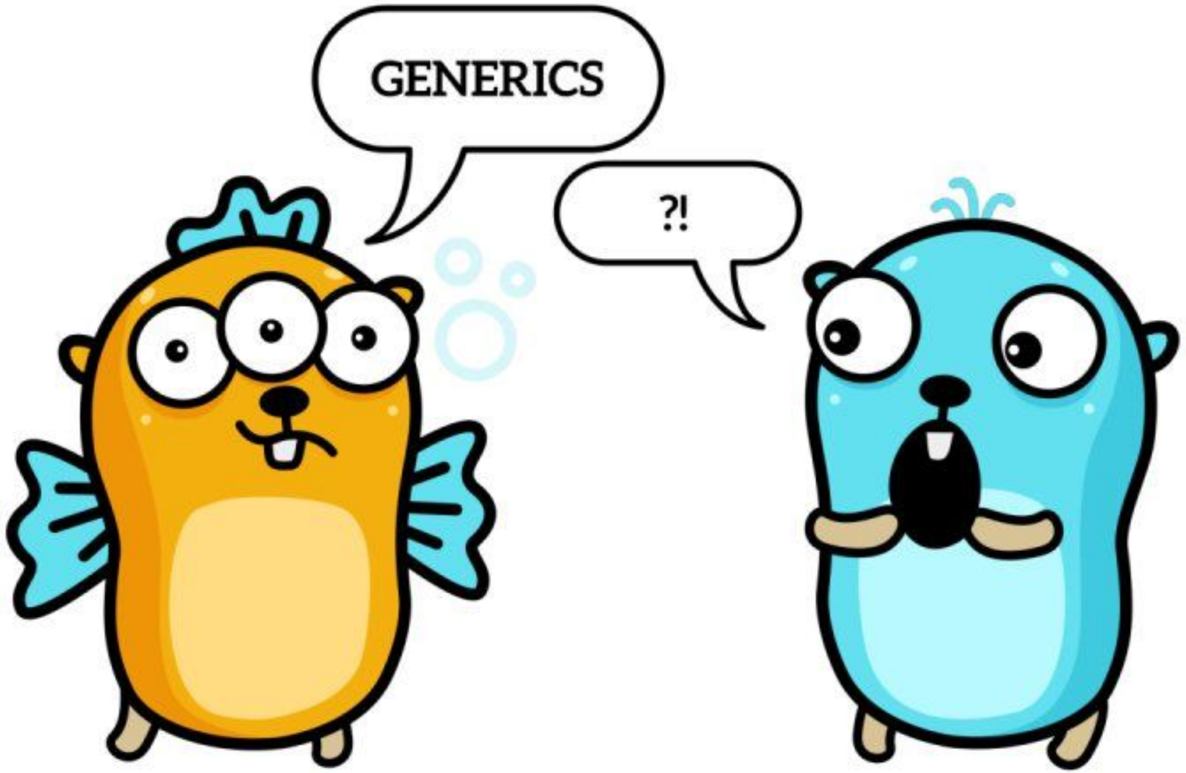

GENERICS
?!

We don’t need generics!

What generics alternatives do we have?

What generics alternatives do we have?

➜ Manual monomorphization

What generics alternatives do we have?

➜ Copy => Paste

What generics alternatives do we have?

✓ Copy => Paste

➜ Interfaces

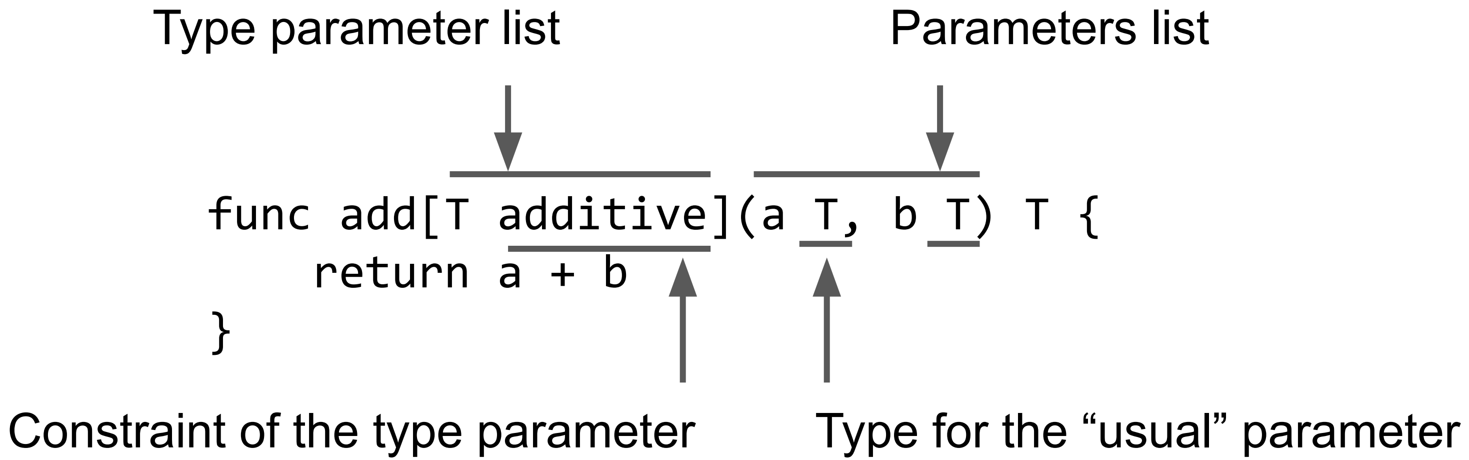
What generics alternatives do we have?

- ✓ Copy => Paste
- ✓ Interfaces
- ➜ Reflection

What generics alternatives do we have?

- ✓ Copy => Paste
- ✓ Interfaces
- ✓ Reflection
- ➜ Code generation

Syntax



Type parameter list
Parameters list
Constraint of the type parameter
Type for the “usual” parameter

```
func add[T additive](a T, b T) T {
    return a + b
}
```

Unconstrained generics

Unconstrained generics

```
func last[T any](a []T) T {
    return a[len(a)-1]
}
```

Unconstrained generics

```
func last[T any](a []T) T {
    return a[len(a)-1]
}

// any is an alias for interface{}
```

Unconstrained generics

```
func last[T any](a []T) T {
    return a[len(a)-1]
}

// any is an alias for interface{}

func less[T any](a T, b T) bool {
    return a < b
}
```

Unconstrained generics

```
func last[T any](a []T) T {
    return a[len(a)-1]
}

// any is an alias for interface{}

func less[T any](a T, b T) bool {
    return a < b // Error
}

// you can not compare any type
```

Unconstrained generics

```
func last[T any](a []T) T {
    return a[len(a)-1]
}

// any is an alias for interface{}

func less[T any](a T, b T) bool {
    return a < b // Error
}

// you can not compare any type
```

```
func less[T comparable](a T, b T) bool {
    return a < b
}

// comparable is a build in constraint
```

You can constraint generics by type

You can constraint generics by type

```
type additive interface {
    int|uint|~float64
}

func add[T additive](a T, b T) T {
    return a + b
}
```

You can constraint generics by type

```
type additive interface {
    int|uint|~float64
}

func add[T additive](a T, b T) T {
    return a + b
}

// Constraint literal
func add[T int|~uint](a T, b T) T {
    return a + b
}
```

You can constraint generics by type

```
type additive interface {
    int|uint|~float64
}

func add[T additive](a T, b T) T {
    return a + b
}

// Constraint literal
func add[T int|~uint](a T, b T) T {
    return a + b
}
```

```
type additive interface {
    int|uint|~float64
}
```

$\uparrow$

What’s this?

We have a new token! ~

We have a new token! ~

~T denotes the set of types whose underlying type is T

We have a new token! ~

~T denotes the set of types whose underlying type is T

```
type additive interface {
    int|uint|~uint64
}

type myUint uint64

func add[T additive](a T, b T) T {
    return a + b
}

x, y := myUint(0), myUint(1.18)
z := add(x, y)
```

You can constraint generics by interface

You can constraint generics by interface

```
type Stringer interface {
     String() string
}

func Tos[T Stringer](s []T) []string {
   var ret []string
   for _, v := range s {
      ret = append(ret, v.String())
   }

   return ret
}
```

Generics in structures

Generics in structures

```
// List is a doubly-linked list.
type List[V any] struct {
    Front, Back *Node[V]
}

// Node is a node in the linked list.
type Node[V any] struct {
    Value      V
    Prev, Next *Node[V]
}
```

Generics in structures

```go
// List is a doubly-linked list.
type List[V any] struct {
    Front, Back *Node[V]
}

// Node is a node in the linked list.
type Node[V any] struct {
    Value      V
    Prev, Next *Node[V]
}

func main() {
    myList := List[int]{}
}
```

Generics in structures

```go
// List is a doubly-linked list.
type List[V any] struct {
    Front, Back *Node[V]
}

// Node is a node in the linked list.
type Node[V any] struct {
    Value      V
    Prev, Next *Node[V]
}

func main() {
    myList := List[int]{}
}
```

```go
func last[T any](a []T) T {
    return a[len(a)-1]
}

func main() {
    xx := []int{1,2,3}
    print(last[int](x))
}
```

Generics in structures

```go
// List is a doubly-linked list.
type List[V any] struct {
    Front, Back *Node[V]
}

// Node is a node in the linked list.
type Node[V any] struct {
    Value      V
    Prev, Next *Node[V]
}

func main() {
    myList := List[int]{}
}
```

```go
func last[T any](a []T) T {
    return a[len(a)-1]
}

func main() {
    xx := []int{1,2,3}
    print(last(x))
}
```

Benchmarks

Benchmarks: general speed

Go 1.18 (generic)

```
func ContainsG[T comparable](s []T, e T)
bool {
     for _, a := range s {
          if a == e {
               return true
          }
     }
     return false
}
```

Benchmarks: general speed

Go 1.18 (generic)

```go
func ContainsG[T comparable](s []T, e T)
bool {
	for _, a := range s {
		if a == e {
			return true
		}
	}
	return false
}
```

Go 1.17 (reflect)

```go
func ContainsR(in interface{}, elem interface{}) bool{
	inValue := reflect.ValueOf(in)
	if inValue.Type().Kind() != reflect.Slice {
		panic("'in' is not a Slice")
	}
	for i := 0; i < inValue.Len(); i++ {
		if equal(elem, inValue.Index(i)) {
			return true
		}
	}
	return false
}

func equal(e interface{}, val reflect.Value) bool {
	if val.IsZero() {
		return val.Interface() == e
	}
	return reflect.DeepEqual(val.Interface(), e)
}
```

Benchmarks: general speed

```
const l = 1000

func Benchmark____(b *testing.B) {
	s := make([]int, l)
	for i := 0; i < l; i++ {
		s[i] = i
	}

	for n := 0; n < b.N; n++ {
		Contains___(s, l-1)
	}
}
```

Benchmarks: general speed

```
const l = 1000

func Benchmark____(b *testing.B) {
	s := make([]int, l)
	for i := 0; i < l; i++ {
		s[i] = i
	}

	for n := 0; n < b.N; n++ {
		Contains___(s, l-1)
	}
}
```

```
[0] $ go test -bench=.
goos: linux
goarch: amd64
cpu: Intel(R) Core(TM) i5-8365U CPU

Reflect     19527    64353   ns/op
Generic   3909652      292.7 ns/op
Native    3977557      307.3 ns/op
```

Benchmarks: general speed

```
func Benchmark__(b *testing.B) {
     for n := 0; n < b.N; n++ {
          Fib__(20)
     }
}

func Fib_(a T)T {
     if a <= 1 {
          return a
     }
     return Fib_(a-1) + Fib_(a-2)
}
```

Benchmarks: general speed

```go
func Benchmark__(b *testing.B) {
	for n := 0; n < b.N; n++ {
		Fib__(20)
	}
}

func Fib_(a T)T {
	if a <= 1 {
		return a
	}
	return Fib_(a-1) + Fib_(a-2)
}
```

```
[0] $ go test -bench=.
goos: linux
goarch: amd64
cpu: Intel(R) Core(TM) i5-8365U CPU

Interface    13354      88085 ns/op
Generic      39980      32544 ns/op
Native       37729      32275 ns/op
```

Benchmarks: compilation time

Benchmarks: compilation time

Go 1.18 (generic)

```
type number interface {
     ~int | ~int32 | ~int64 | ~float32 |
     ~float64 | ~uint | ~uint64
}

func MaxGeneric**[T number](a, b T) T {
     if a > b {
          return a
     }

     return b
}
```

Benchmarks: compilation time

Go 1.18 (generic)

```
type number interface {
     ~int | ~int32 | ~int64 | ~float32 |
     ~float64 | ~uint | ~uint64
}

func MaxGeneric**[T number](a, b T) T {
     if a > b {
          return a
     }

     return b
}
```

Go 1.17 (native)

```
func MaxInt**(a, b int) int {
     if a > b {
          return a
     }

     return b
}
```

Benchmarks: compilation time

1. Without actual function calls

Benchmarks: compilation time

1. Without actual function calls

Static typing

```
[0] $ time go build -o ogo static.go dummy_main.go
real 0m1.347s
user 0m3.566s
sys  0m0.192s
```

Benchmarks: compilation time

1. Without actual function calls

Static typing

```
[0] $ time gotip build -o ogo static.go dummy_main.go
real 0m1.347s
user 0m3.566s
sys  0m0.192s
```

Generics

```
[0] $ time gotip build -o ogo generics.go dummy_main.go
real 0m0.499s
user 0m0.843s
sys  0m0.097s
```

Benchmarks: compilation time

2. With functions calls

Benchmarks: compilation time

2. With functions calls

Static typing

```
[0] $ time go build -o ogogo static.go calls_main.go
real 0m5.614s
user 0m9.924s
sys  0m0.423s
```

Generics

```
[0] $ time build -o ogogo generics.go calls_main.go
real 0m5.419s
user 0m10.395s
sys  0m0.409s
```

What else we need to know about performance?

What else we need to know about performance?

Generics are like monomorphization

What else we need to know about performance?

Generics are like monomorphization, but not completely.

What else we need to know about performance?

Generics are like monomorphization, but not completely.

Built in types and structs

What else we need to know about performance?

Generics are like monomorphization, but not completely.

Built in types and structs

- Function is generated per every type

What else we need to know about performance?

Generics are like monomorphization, but not completely.

Built in types and structs

- Function is generated per every type

Interfaces, pointers to interfaces and <u>pointers to structs</u>

What else we need to know about performance?

Generics are like monomorphization, but not completely.

Built in types and structs

- Function is generated per every type

Interfaces, pointers to interfaces and <u>pointers to structs</u>

- Function is generated per every “GCShape”

What else we need to know about performance?

Generics are like monomorphization, but not completely.

Built in types and structs

- Function is generated per every type

Interfaces, pointers to interfaces and <u>pointers to structs</u>

- Function is generated per every "GCShape"
- We will have GCShape dictionary

What else we need to know about performance?

Generics are like monomorphization, but not completely.

Built in types and structs

- Function is generated per every type

Interfaces, pointers to interfaces and pointers to structs

- Function is generated per every "GCShape"
- We will have GCShape dictionary
- Every function call will look up for a function in the dictionary

What else we need to know about performance?

```
name                       time/op       allocs/op
Monomorphized-16           5.06µs ± 1%  2.00 ± 0%
Iface-16                   6.85µs ± 1%  3.00 ± 0%
GenericWithPtr-16          7.18µs ± 2%  3.00 ± 0%
GenericWithExactIface-16  9.68µs ± 2%  3.00 ± 0%
GenericWithSuperIface-16  17.6µs ± 3%  3.00 ± 0%
```

What else we need to know about performance?

```
name                         time/op       allocs/op
Monomorphized-16             5.06µs ± 1%   2.00 ± 0%
Iface-16                     6.85µs ± 1%   3.00 ± 0%
GenericWithPtr-16            7.18µs ± 2%   3.00 ± 0%
GenericWithExactIface-16     9.68µs ± 2%   3.00 ± 0%
GenericWithSuperIface-16     17.6µs ± 3%   3.00 ± 0%
```

Generics can make your Go code slower

Generics via Dictionaries and Gcshape Stenciling

When to use/not to use generics?

When to use/not to use generics?

Use generics to not repeat yourself.

When to use/not to use generics?

Use generics to not repeat yourself.

As a conclusion:

try not to use generics
until you start to repeat yourself.

Read more:

Generics Proposal

Repository with Go generics research and examples:

Go generics: the hard way

Talk from masters of Go:

GopherCon: R Griesemer & Ian Lance Taylor

Functional library with Filter, Map, Reduce, etc.:

github.com/samber/lo

Benchmarks code from these slides

That’s all!

Questions?