



Generics

Discussion Points

- Introduction to generics in Go





Introduction to generics in Go

What is Generics?

Generics allow you to write code without specifying the exact data types for inputs or outputs — meaning you can create functions or data structures without defining the type of values they handle. This approach, known as parametric polymorphism, enables more flexible and reusable code.

That wasn't available prior to **Go 1.18**?

Technically, there were a few ways of dealing with “generics” in Go that were introduced even before Go generics was released:

- Using interface with switch statement and type casting
- Making use of the reflection package with arguments validation

However, compared to official Go generics, these approaches are insufficient.

- Lower performance when using types switch and casting
- Higher loss of type safety: Interface and reflection are not type-safe which means the code may pass any type that can cause panic during run time when it goes unnoticed during compilation
- Poor code readability
- Lack of custom-derived types support





Introduction to generics in Go

What is Generics?

Generics are a valuable feature in Go because they offer several benefits and address common programming challenges. Here are some reasons why generics are important

- Code Reusability
- Type Safety
- Improved Readability
- Performance Optimization
- Avoiding Redundant Code

Type parameter Constraints



```
func Last[T any](s []T) T {  
    return s[len(s)-1]  
}
```





Introduction to generics in Go

What are Type Parameters?

In Go, type parameters are defined using a type parameter list enclosed in square brackets right after the function, data structure, or interface name. These type parameters are typically represented by single uppercase letters or a sequence of uppercase letters.

```
func print[T any](v T) {  
    fmt.Println(v)  
}  
  
func keys[K comparable, V any](m map[K]V) []K {  
    // creating a slice of type K with length of map  
    ...  
    return key  
}
```

```
type MyStruct[T any] struct {  
    inner T  
}  
  
func (m *MyStruct[T]) Get() T {  
    return m.inner  
}  
  
func (m *MyStruct[T]) Set(v T) {  
    m.inner = v  
}  
  
func main() {  
    s := MyStruct[int]{5}  
    fmt.Println(s.Get())  
}
```





Introduction to generics in Go

What are Type Constraints?

Type constraints in generics define which types can be used with a generic function or data structure, ensuring that only compatible types are utilized and enforcing type safety at compile time. These constraints are defined using the interface keyword, followed by the interface name and the methods that a type must implement. For instance, consider the following generic function that uses a type constraint:

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

In this example, the type parameter "T" is constrained by the "**comparable**" interface, which requires that the type **implements the comparison operators (>, <, >=, <=)**. This ensures that the function can only be called with types that support comparison.





Introduction to generics in Go

Create own Type Constraints?

Any type can be used as a type constraint.

```
type Number interface {  
    int | int16 | int32 | int64  
}
```

This means an interface may not only define a set of methods, but also a set of types.

The above interface constraint allows types of int, int16, int32, and int64. The types are the constraint union, which is symbolized by the pipe character | which separates the types.





Introduction to generics in Go

Type approximation?

Go allows creating user-defined types from predefined types like `int`, `string`, etc. `~` operators allow us to specify that interface also supports types with the same underlying types.

```
type Number interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64 | ~float32 | ~float64
}

type Point int

func Min[T Number](x, y T) T {
    if x < y {
        return x
    }
    return y
}

func main() {
    x, y := Point(5), Point(2)

    fmt.Println(Min(x, y))
}
```



Thank you. Questions?

“Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.”

- **Albert Einstein**

