MARCH 23, 2021 / #GO

Generics in Go Explained with Code Examples

Pramono Winata

Generics were proposed a few years ago for Go, and they have finally been accepted into the language earlier this year. And they're scheduled to be officially released at the end of this year.

How will Generics really affect Go? Will it change how we code?

To really answer these questions, we will need to take a look at how generics work. Conveniently, the devs have provided us with a web compiler where we can experiment with generics ourselves.

What Do Generics Really Change in Go?

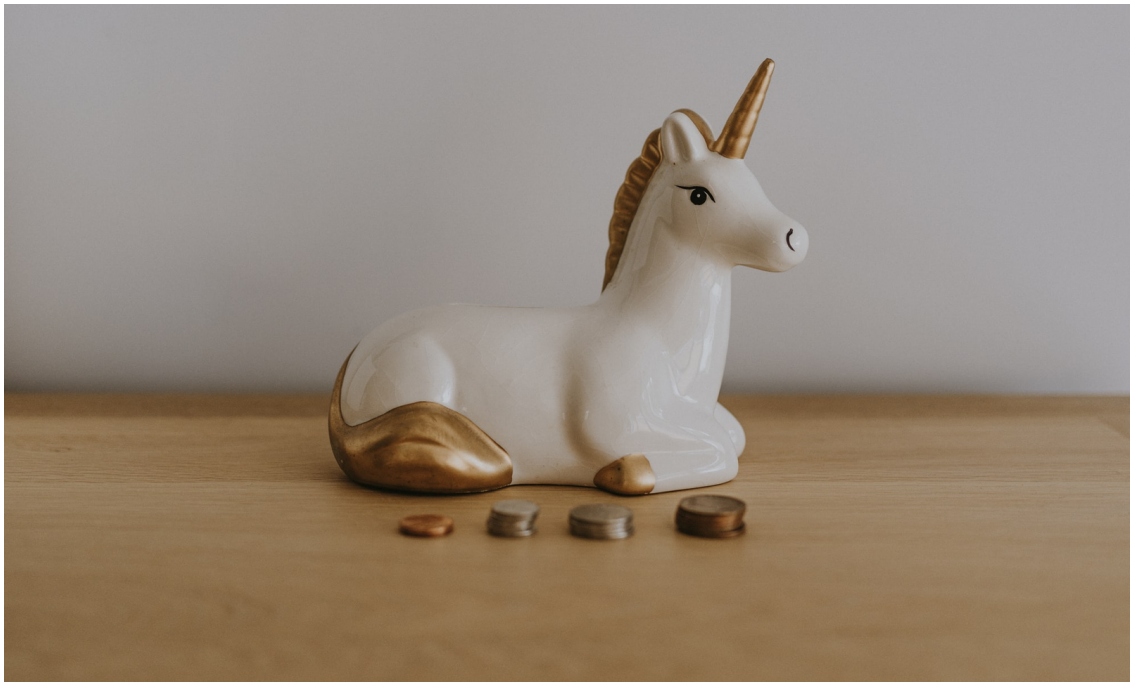

Photo by Annie Spratt / Unsplash

Generics allow our functions or data structures to take in several types that are defined in their generic form.

To truly understand what this means, let's take a look at a very simple case.

Let's say you need to make a function that takes one slice and prints it. Then you might write this type of function:

```
func Print(s []string) {
    for _, v := range s {
        fmt.Print(v)
    }
}
```

Simple, right? What if we want to have the slice be an integer? You will need to make a new method for that:

```
func Print(s []int) {
        for _, v := range s {
                fmt.Print(v)

        }
}
```

These solutions might seem redundant, as we're only changing the parameter. But currently, that's how we solve it in Go without resorting to making it into some interface.

And now with generics, they will allow us to declare our functions like this:

```
func Print[T any](s []T) {
        for _, v := range s {
                fmt.Print(v)
        }
}
```

In the above function, we are declaring two things:

1. We have T, which is the type of the `any` keyword (this keyword is specifically defined as part of a generic, which indicates any type)
2. And our parameter, where we have variable `s` whose type is a slice of `T` .

We will now be able to call our method like this:

```
func main() {
    Print([]string{"Hello, ", "playground\n"})
```

One method for any type of variable – neat, huh?

This is just one of the very basic implementations for generics. But it looks good so far.

Let's explore more and see how far generics can take us.

Limitations of Generics

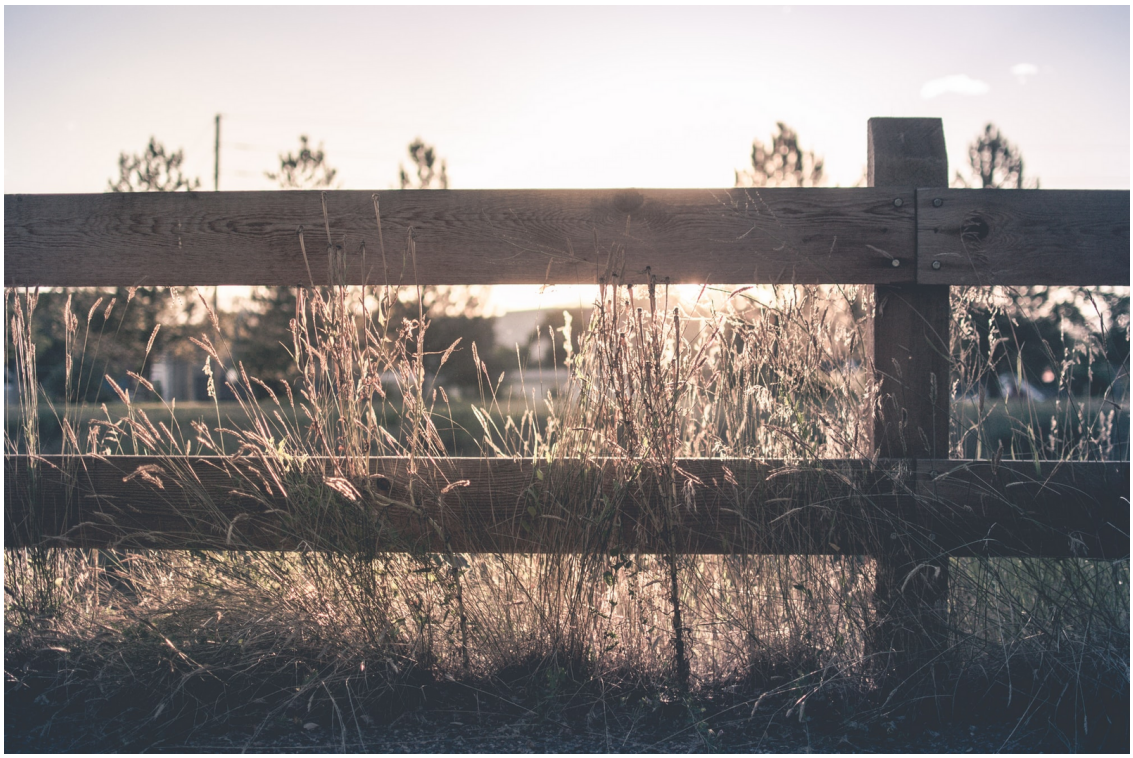

Photo by Nick Tiemeyer / Unsplash

We have seen what generics can do. They let us specify a function that can take in any kind of parameter.

But the example I gave before was a very simple one. There are limitations on how far generics can take us. Printing, for example, is

What if we want to do more complex things? Let's say that we have defined our own methods for a structure and want to call it:

```
package main

import (
    "fmt"
)

type worker string

func (w worker) Work(){
    fmt.Printf("%s is working\n", w)
}


func DoWork[T any](things []T) {
    for _, v := range things {
        v.Work()
    }
}

func main() {
    var a,b,c worker
    a = "A"
    b = "B"
    c = "C"
    DoWork([]worker{a,b,c})
}
```

And you will get this:

```
type checking failed for main
prog.go2:25:11: v.Work undefined (type bound for T has no method
```

It fails to run because the slice processed inside the function is of

We can actually make it work, though, by using an interface:

```
package main

import (
    "fmt"
)

type Person interface {
    Work()
}

type worker string

func (w worker) Work(){
    fmt.Printf("%s is working\n", w)
}

func DoWork[T Person](things []T) {
    for _, v := range things {
        v.Work()
    }
}

func main() {
    var a,b,c worker
    a = "A"
    b = "B"
    c = "C"
    DoWork([]worker{a,b,c})
}
```

And it will print out this:

```
A is working
B is working
C is working
```

The generic works well, too:

```
package main

import (
    "fmt"
)

type Person interface {
    Work()
}

type worker string

func (w worker) Work(){
    fmt.Printf("%s is working\n", w)
}

func DoWorkInterface(things []Person) {
    for _, v := range things {
        v.Work()
    }
}

func main() {
    var d,e,f worker
    d = "D"
    e = "E"
    f = "F"
    DoWorkInterface([]Person{d,e,f})
}
```

This will give us the following result:

```
D is working
E is working
F is working
```

code.

Generics are still in their very early phases of development, and they do still have their limits for doing complex processing.

Playing Around With Constraint

Photo by Paulo Brandao / Unsplash

Earlier, we came upon the `any` type for our generic constraint. Aside from that type, there are several other constraints we can use.

One of the constraints is `comparable`. Let's see how it works:

```
func Equal[T comparable](a, b T) bool {
    return a == b
}
```

Aside from that, we can also try to make our own constraint like this:

```
package main

import(
    "fmt"
)

type Number interface {
    type int, float64
}

func MultiplyTen[T Number](a T) T{
    return a*10
}

func main() {
    fmt.Println(MultiplyTen(10))
    fmt.Println(MultiplyTen(5.55))
}
```

And I think that's pretty neat – we can have one function for a simple mathematical expression. Usually we will end up making two functions to take it in or we'll use reflection so we only write one function.

While this is pretty cool, we'll still need to experiment quite a bit with making our own constraints. It's still too early to know their limitations. And we should be careful not to abuse it and only use it if we are really sure it is needed.

Other Ways to Use Generics

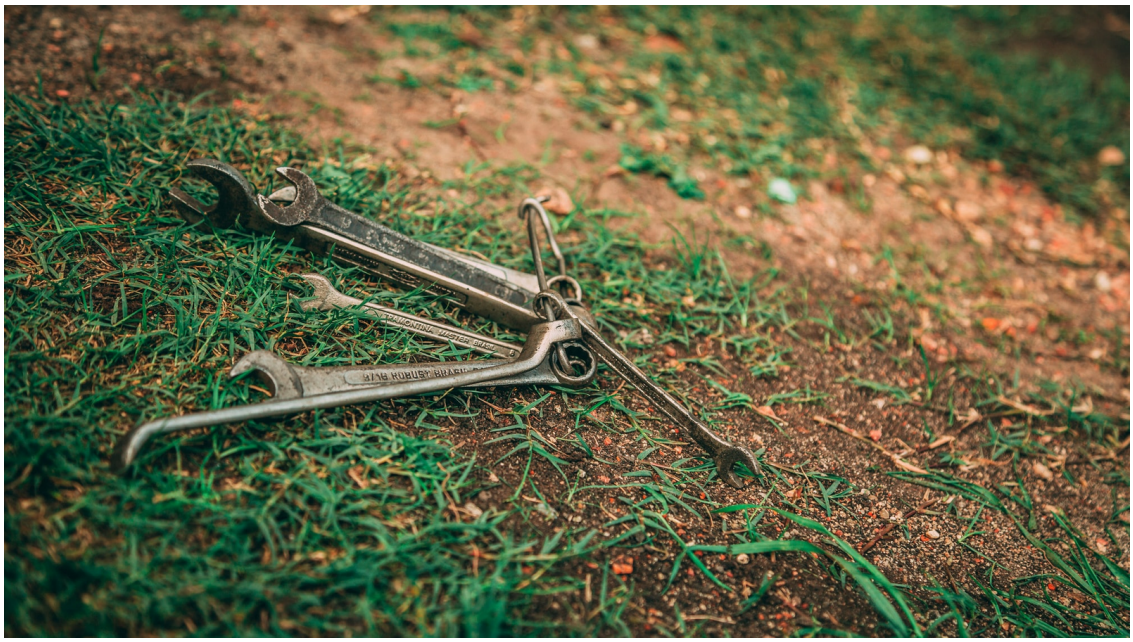

Photo by Marcelo Franchi / Unsplash

Aside from using generics as part of a function, you can also declare them as variables like this:

```
type GenericSlice[T any] []T
```

And you can use this either as a parameter in a function or you can make method out of that type:

```
func (g GenericSlice[T]) Print() {
    for _, v := range g {
        fmt.Println(v)
    }
}

func Print [T any](g GenericSlice[T]) {
```

```
}

func main() {
    g := GenericSlice[int]{1,2,3}

    g.Print() //1 2 3
    Print(g) //1 2 3
}
```

The usage varies depending on your needs. All I can say is that we still need to experiment with generics more to see what use cases work best.

My Take On Generics

Generics are still in their very early phases (they're not even out yet!), but I'm pretty impressed with how they're made. There aren't many complicated terms and libraries needed to implement generics, and this simplicity is great.

There are several use cases where I can already see that using generics will be better (like the case with the multiply method). One thing that a lot of people seem to be confused about is that generics might be a replacement for using interfaces (both interface{} type and Interface implementation).

My advice is not to think of generics as a replacement for anything. Generics are just another tool provided for us in our coding life. Also, Generics might look fancy and cool, and you might want to use them in every block of your code. But don't overuse them – only whenever they're really needed, not whenever they can fit.

And that's it. Thanks for reading my article, and I truly hope generics can become useful for you

when writing this article. It explains a lot of the backstory regarding generics in Go.

Have fun with Generics!

Pramono Winata

Read more posts by this author.

If you read this far, tweet to the author to show them you care.

Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

Trending Guides