KNOW GO GENERICS

"Beautifully explained"
—Pavel Anni

POLYMORPHIC PROGRAMMING IN GO

JOHN ARUNDEL

2. Type parameters

> *I learned very early the difference between knowing the name of something, and knowing something.*
>
> —Richard Feynman, "The World From Another Point of View"

Now that we've described what generic programming is, and how it came to Go, it's time to look at exactly what Go's generics support involves, and what we can do with it.

Generic functions

And now that's easier to explain: it means we can write functions like `PrintAnything` and `AddAnything`!

Introducing T, the arbitrary type

To be precise, we can write *generic functions* that take parameters not of some specific named type, but of some arbitrary type that we don't have to specify in advance. Let's call it T, for "type".

When the function is actually *called* in our program, T will be some specific type, such

as `int`. But we don't want to have to specify that in advance when we're writing the function, so we're just going to use T as a placeholder for whatever the type ends up being.

In fact, there might be more than one T in our finished program. For example, we might call our generic function with a `float64` value, in which case T will be `float64`. But elsewhere we might call the same function with a `string` value, in which case T will be `string`.

This T placeholder is called a *type parameter*. A generic function in Go, then, is a function that takes a type parameter. So what does that look like?

Let's take the simplest imaginable case first. We'll write a `PrintAnything` function with a type parameter T, where T is a placeholder for any type:

```
func PrintAnything[T any](v T) {
```

Type parameters

What does this mean? For any type T, `PrintAnything[T]` takes a T parameter (that is, a parameter whose type is T), and returns nothing. The function signature just says that in Go instead of English.

While `v` is just an ordinary parameter, of some unspecified type, T is different. T is a new kind of parameter in Go: a *type parameter*.

We say that `PrintAnything` is a *parameterised* function, that is, a generic function *on* some type T. For short, we usually just talk about `PrintAnything[T]`, pronounced "`PrintAnything` *of* T".

Instantiation

What type *is* T, specifically? It depends what we decide to pass to the function when it's called.

Suppose we want to pass it an `int` value, for example:

```
var x int = 5
PrintAnything(x)
// Output:
// 5
```

The compiler is smart enough to know that `x` is an `int`, and therefore it needs to compile (and call) a version of `PrintAnything[T]` where T is `int`.

This is called *instantiating* the function, as in "creating an instance of it". In effect, the generic `PrintAnything` function is like a kind of template, and when we call it with some specific type, we create a specific *instance* of the function that takes that type—for example, `int`.

What if we have another call to `PrintAnything[T]` somewhere else in the program, and this time T is a different type, such as `string`? Well, that's okay. The compiler will produce another version of `PrintAnything`, this time one that takes a string argument.

In most cases, as in these examples, the compiler can *infer* the type of T from the value that's supplied at the site of the function call. When that isn't possible, the compiler will let you know.

For example, suppose we have a generic function `Something[T any]` that *returns* a value of T. And suppose we call it like this:

```
x := Something()
```

What is T here? We don't know, and nor does the compiler, so it must complain:

```
cannot infer T
```

All is not lost, however. We can specify which particular T we want in this case by putting it inside square brackets after the function name:

```
x := Something[int]()
```

It's always okay to *explicitly instantiate* a parameterised function or type in this way. But you only *have* to do it when the compiler can't automatically infer the required type, which isn't all that often.

Stencilling

This approach to implementing generics is sometimes called *stencilling*, which is a rather apt name. You can imagine the compiler spray-painting a bunch of similar versions of the function, differing only in the type of the parameter they take.

We could have done the same thing ourselves using the existing *code generation* machinery in Go, and indeed many people did do exactly that before the introduction of generics.

This makes for efficient machine code, because there's no indirection (unlike with interface values). We don't need type assertions, because each different implementation of `PrintAnything` knows exactly what concrete type it's getting.

This isn't a particularly compelling example of generics in action, though, because it was already possible to write `PrintAnything` using a method-set interface: `any`. We could just pass the argument straight to `fmt.Println`, which also takes `any`.

Getting started

Let's look at a slightly more interesting, though still rather contrived, example.

An `Identity` function

Suppose we want to write a function called `Identity` that simply returns whatever value you pass it. (I know it doesn't sound very useful, but bear with me.) How could we write such a function in Go, without having to implement a separate version for each possible type of value?

This is where we start to go beyond the limits of interfaces. Using `any`, for example, we'd have to write something like:

```
func Identity(v any) any {
    return v
}
```

This works, but it isn't really satisfactory. As we saw with `AddAnything` in the previous chapter, we don't have any way to tell the compiler that the function's parameter and its result must be the *same* concrete type, whatever it is. And clearly that needs to be the case here: if we pass this function an `int`, we expect to get an `int` back.

Now we know how to specify that requirement, using a type parameter:

```
func Identity[T any](v T) T {
    return v
}
```

Notice that, although it looks similar to the non-generic version, there's an important difference. Whatever T turns out to be (and it can be any type), *both* the parameter and the function's result must be of that type.

Instantiating `Identity`

Suppose we call this function somewhere in our program with a string argument, then:

```
fmt.Println(Identity("Hello"))
// Output:
// Hello
```

You now know how this works. Under the hood, the compiler instantiates a version of `Identity` that takes a string parameter and returns a string result. This is just a plain, ordinary Go function that we could have written ourselves, or generated mechanically.

The point is, of course, that *we* don't need to supply a separate version of `Identity` for each concrete type that we want to use. Instead, we just write it once for some arbitrary type T, and the compiler will automatically generate a version of `Identity` for each type that's actually used in our program.

Exercise: Hello, generics

Now it's over to you to write your first generic function in Go! Let's work through it together, step by step.

First of all, make sure you've checked out a copy of the GitHub repo for this book:

- https://github.com/bitfield/kg-generics2

Open the `exercises/print` folder in your code editor and take a look at the `print_test.go` file.

You'll find this test:

```
func TestPrintAnythingTo_PrintsToGivenWriter(t *testing.T) {
    t.Parallel()
    buf := new(bytes.Buffer)
    print.PrintAnythingTo(buf, "Hello, world")
    want := "Hello, world\n"
    got := buf.String()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing `exercises/print`)

Running the test

Run the test using your editor, or the `go test` command. You'll see that right now the test doesn't compile, because the required function doesn't exist:

```
undefined: print.PrintAnythingTo
```

Remember, you'll need at least Go version 1.18 to be able to use generics, including running this test and implementing the function to make it pass. That's because we're using some new syntax that doesn't exist in Go version 1.17 and earlier.

If you try to compile this code with an older version of Go that doesn't support generics, you'll get a rather confusing additional error message:

```
type string is not an expression
```

So if you see this, you need to upgrade your Go. Follow the instructions in the introduction to this book to do that. Now read on!

GOAL: Get the test passing!

HINT: To make this test even compile, you'll need to define a generic function in the `print` package named `PrintAnythingTo` that takes one parameter of type `io.Writer`, and another value that's of some unspecified type.

In other words, `PrintAnythingTo` has a type parameter we'll refer to as T, which can be any type, and it takes a parameter of this type T, just like `Identity`. But unlike `Identity`, it also takes another parameter, which is of type `io.Writer`.

To make the test *pass*, your function will need to write the supplied value to the supplied writer. It's up to you how to do this, but you might like to use `fmt.Fprintln`, like our `PrintTo` example in the previous chapter.

The necessary `go.mod` and `print.go` files are already set up for you. All you need to do is edit `print.go` and add the `PrintAnythingTo` function, then run the test again.

SOLUTION: Here's my suggested solution; it's fine if yours looks different, so long as it passes the test. It's just for comparison, or if you need a little extra help on this tricky first exercise.

```
package print

import (
    "fmt"
    "io"
)

func PrintAnythingTo[T any](w io.Writer, p T) {
    fmt.Fprintln(w, p)
}
```

(Listing `solutions/print`)

We use the `[T any]` syntax to say that `PrintAnything` is about some arbitrary type T, and it takes a parameter—p, the thing to print—of that type, whatever it actually turns out to be. In the test, as it happens, that's a string, but it *could* have been any type. If you like, you can add more tests that call `PrintAnything` with different types, such as `int` or `float64`.

Composite types

So far, we've figured out how to define a generic function that, for some type T, takes a parameter of type T:

```
func Identity[T any](v T) {
```

So is that it? Are we restricted to declaring only parameters of type T itself, or could we also take some *composite* type? That is, some type *involving* T, not just T itself? For example, a *slice* of T?

Slices of some arbitrary type

We could indeed. Suppose we wanted to write a function `Len` that returns the length of a given slice. And its parameter will always be a slice of *something*: that is, of some arbitrary element type. Let's call it E, for "element".

The signature of `Len`, then, might look something like this:

```
func Len[E any](s []E) int {
```

It's conventional, though not required, to capitalise the names of type parameters. Those names can be whatever you like, but again it's conventional to use a single letter: T for any type, E for an element type of a slice, and so on.

Other generic composite types

But we can also use a type parameter in other kinds of composite type. For example, we can write a generic function on a *channel* of some element type E:

```
func Drain[E any](ch <-chan E) {}
```

We could even write a *variadic* function: that is, a function that takes a variable number of arguments. In this case, it would take a variable number of channels of E:

```
func Merge[E any](chs ...<-chan E) <-chan E {
```

Actually implementing these functions is a topic for another book (stay tuned), but I'm sure you can see the possibilities.

Generic types

Generic functions are great, but we can do more. We can also write generic *types*. What does that mean?

We often deal with *collections* of values in Go. For example, consider this slice type:

```
type SliceOfInt []int
```

You already know that, just as an `int` variable can only hold `int` values, a `[]int` slice can only hold `int` elements. We wouldn't want to have to also define `SliceOfFloat`, `SliceOfString`, and so on.

Defining a generic slice type

Could we write a *generic* type definition that takes a type parameter, just like a generic function? For example, could we make a slice of any type?

Yes, we could:

```
type Bunch[E any] []E
```

Just as we could define a function such as `Len` that takes a slice of an arbitrary element type, we've now given a name to a new *type*: a slice of `E`, for some type `E`.

And just as with generic functions, a generic type is always *instantiated* on some specific type when it's used in a program. That is to say, a `Bunch[int]` will be a slice of `int`, a `Bunch[string]` will be a slice of strings, and so on.

Each of these is a distinct concrete type, as you'd expect, and we can write a `Bunch` literal by giving the type we want in square brackets:

```
b := Bunch[int]{1, 2, 3}
```

The elements all have the same type

It's very important to understand that, even though a `Bunch` is defined as a slice of `E` for any type `E`, any *particular* `Bunch` can't contain values of *different* types. All the elements of a `Bunch[T]` must be of type `T`, whatever it is.

For example, we can't create a `Bunch` of one type, and then try to append a value of a different type:

```
b := Bunch[int]{1, 2, 3}
b = append(b, "hello")
// cannot use "hello" (untyped string constant) as int value in
// argument to append
```

We can have a `Bunch[int]` *or* a `Bunch[string]` *or* a `Bunch` of elements of any other specific type. What we can't have is a `Bunch` of *mixed-type* elements.

It's easy to hear a term like "generic slice" and jump to the conclusion that it means "slice containing values of different types". But that's actually not the case.

Generic types need to be instantiated

There are, in a sense, *no generic types in Go*. I know that sounds crazy, but stick with me.

That is, you can *define* generic types like `Bunch[E]`, but to actually use them in your program, you need to *instantiate* them on some specific type, like `int`.

At that point, what you have is an ordinary Go slice of `int`, and it stands to reason that such a slice can only contain `int` elements.

So just because we used a type parameter, it doesn't mean we can create a single slice that contains elements of different types. What we can create are different *slice types*, such as `[]int`, or `[]string`, without having to specify their element type in advance.

One way to express this is to say that, while we can define generic types at compile time, there are no generic types *at run time*.

There's one partial exception to this, which you're already familiar with: interface types. We could always create a slice of `any` in Go, and populate it with elements of different dynamic types. But, for the reasons we've discussed, this isn't the same thing as a truly generic type.

Exercise: Group therapy

Over to you again now, to try your hand at the `group` exercise. This time, you'll need to define a generic slice type `Group[E]` to pass the following test:

```
func TestGroupContainsWhatIsAppendedToIt(t *testing.T) {
    t.Parallel()
    got := group.Group[string]{}
    got = append(got, "hello")
    got = append(got, "world")
    want := group.Group[string]{"hello", "world"}
    if !slices.Equal(want, got) {
        t.Errorf("want %v, got %v", want, got)
    }
}
```

(Listing `exercises/group`)

GOAL: Get the test passing!

HINT: Well, a "group" is a lot like a "bunch", and I don't think you'll need any more hints than that. There's no trick here: it's just a confidence builder to get you used to defining generic types.

SOLUTION: And here's what that looks like, just as we saw before with the `Bunch` type:

```
type Group[E any] []E
```

(Listing `solutions/group`)

I told you it was easy!

Generic function types

You probably know that *functions are values* in Go. That is, you can pass a function as an argument to another function, you can *return* a function from a function, and you can declare variables or struct fields of a particular function type.

In other words, just as Go values can be integers, floats, or strings, they can also be functions. For example, a function that takes an `int` parameter and returns `bool` would have the type `func(int) bool`. This is a very handy feature of Go, and we use it a lot.

Generic functions as values

So what if that function were generic? For example, the `Identity[T]` function in our earlier example. How would we declare a variable to which we could assign the function `Identity[T]`? What would be the type of such a variable?

Well, you now know that there's actually no such function as `Identity[T]`, only one or more *instantiations* of that function on a specific type, such as `string`. As we've seen, the `Identity[T]` function definition is just a kind of "stencil" that the compiler uses to produce *actual* functions.

So there *is* such a function as `Identity[string]`, and accordingly we can use it as a value. For example, we can assign it to a variable:

```
f := Identity[string]
```

The type is always instantiated

What is the type of the variable `f`, then? Well, it's whatever the type of `Identity[string]` is. Here's the signature of the generic `Identity` function again:

```
func Identity[T any](v T) T {
```

So if T is `string` in this case, then we feel the type of `Identity[string]` should be:

```
func(string) string
```

Let's ask Go itself. Conveniently, the `fmt` package can report the type of a value, using the `%T` verb. So we'll see what type it thinks `f` is in this case:

```
f := Identity[string]
fmt.Printf("%T\n", f)
// Output:
// func(string) string
```

How straightforward!

There are no generic functions

Just as with generic types, then, *there are no generic functions in Go*, if you want to be a smart-alec about it (and I do).

Any generic functions you may write will in fact be instantiated on some specific type at compile time, and they will then just be plain old functions, like `func(string) string`. If we wanted to give such a specific function type a name, we could:

```
type Stringulator func(string) string
```

Could we define a *generic function type*, though? That is, give a name to the type of a generic function on some arbitrary type T.

For example, could we write something like:

```
type idFunc func[T any](T) T
// syntax error: function type must have no type parameters
```

Nope. If you think about it, how could the compiler compile this? It couldn't, because *all type arguments must be known at compile time*.

What we *can* write is this:

```
type idFunc[T any] func(T) T
```

This is fine, because it's a *generic type*, just like the ones we've already seen. For some T, we're saying, an `idFunc[T]` is a function that takes a T and returns a T.

If this is ever instantiated in our program, it will become some specific type like `func(int) int`. If not, the compiler can safely ignore it. Either way, no unknown types are involved.

Generic types as function parameters

We can create both generic functions and generic types, as we've seen. So an interesting question occurs: could we write a generic function that takes a *parameter* of a generic type?

The answer is absolutely yes:

```
func PrintBunch[E any](v Bunch[E]) {
    fmt.Println(v)
}

func main() {
    b := Bunch[int]{1, 2, 3}
    PrintBunch(b)
    // Output:
```

```
    // [1, 2, 3]
}
```

This is pretty exciting stuff! But there's a limit to what we can do with generic functions that take literally *any* type. We'll see what that limit is in a moment, but first, let's do a little more confidence building.

Exercise: Lengthy proceedings

Now it's your turn to write a generic function on a generic type, to solve the `length` exercise.

```
func TestLenOfSliceIs2WhenItContains2Elements(t *testing.T) {
    t.Parallel()
    s := []int{1, 2}
    want := 2
    got := length.Len(s)
    if want != got {
        t.Errorf("Len(%v): want %d, got %d", s, want, got)
    }
}
```

(Listing `exercises/length`)

GOAL: Your task is to implement a generic function `Len[E]` that returns the length of a given slice. As usual, the test will tell you when you've got it right!

HINT: Actually, the *implementation* is easy, isn't it? We don't need to write code to figure out the length of some slice; there's a built-in function for that (see if you can guess which one).

The tricky bit, if it is tricky, might be writing the *signature* of the `Len` function. But it's not really any more complicated than what we've seen already.

For example, the `PrintBunch` function takes a `Bunch[E]` for some arbitrary element type `E`. Well, so does this one! Can you see what to do?

SOLUTION: Here's one possible answer:

```
func Len[E any](s []E) int {
    return len(s)
}
```

(Listing `solutions/length`)

Constraining type parameters

We saw in the previous chapter that one of the limitations of interface types in Go is that we can't use them with operators such as `+`. Remember our `AddAnything` example?

We can't add `any` to `any`

You might be wondering if the same kind of limitation applies to functions parameterised by `T any`, and indeed it does. If we try to write a generic `AddAnything[T any]`, for example, that doesn't work:

```
func AddAnything[T any](x, y T) T {
    return x + y
}
// invalid operation: operator + not defined on x (variable
// of type T constrained by any)
```

The compiler is always right, but it can sometimes express itself in a rather terse way, so let's unpack that error message a little.

It's complaining about this line:

```
return x + y
```

And it's saying:

```
operator + not defined on x
```

To put it another way, the compiler has no way to guarantee that whatever specific type T happens to be when the function is instantiated, that type will work with the + operator.

It *might*, but then again, it might not, because:

```
variable of type T constrained by any
```

Not every type is "addable"

In other words, because `x` can be *any* type, there's no way to guarantee that, when the program runs, `x` will be one of the types that supports the + operator. So this is just the same kind of problem as when we tried to add together two `any` values.

If `x` and `y` were some struct type, for example, that certainly wouldn't work: structs don't support +, because it's not meaningful to add two structs together. The + operator isn't *defined* on structs, we say.

The compiler is telling us that we can't write the expression `x + y` with values of literally *any* type `T`: that's too broad a range of possible types.

It might be that, in a given program, we only ever instantiate `AddAnything` on types that *do* happen to support `+`, such as `int` or `string`. But that's not good enough for the compiler: we need to *guarantee* that it can't be instantiated on some inappropriate type.

To do this, we need to *constrain* T a little more. That is, to restrict the allowed possibilities for T to only those types that support the operator we want to use. And we'll see how to do that in the next chapter.