CHAPTER 15
Welcome to the Future: Generics in Go

Despite the lower priority placed on features, Go isn't a static, unchanging language. New features are adopted slowly, after much discussion and experimentation. Since the initial release of Go 1.0, there have been significant changes to the patterns that define idiomatic Go. The first was the adoption of the context in Go 1.7. This was followed by the adoption of modules in Go 1.11 and error wrapping in Go 1.13.

The next big change has arrived. Version 1.18 of Go includes an implementation of type parameters, which are colloquially referred to as generics. In this chapter we'll explore why people want generics, what Go's implementation of generics can do, what generics can't do, and how they might change idiomatic patterns.

Generics Reduce Repetitive Code and Increase Type Safety

Go is a statically typed language, which means that the types of variables and parameters are checked when the code is compiled. Built-in types (maps, slices, channels) and functions (such as `len`, `cap`, or `make`) are able to accept and return values of different concrete types, but until Go 1.18, user-defined Go types and functions could not.

If you are familiar with dynamically typed languages, where types are not evaluated until the code runs, you might not understand what the fuss is about generics, and you might be a bit unclear on what they are. It helps if you think of them as "type parameters." We are used to writing functions that take in parameters whose values are specified when the function is called. In this code, we specify that `Min` takes in two parameters of type `float64` and returns a `float64`:

```go
func Min(v1, v2 float64) float64 {
    if v1 < v2 {
        return v1
    }
```

```
        return v2
    }
```

Similarly, we create structs where the type for the fields is specified when the struct is declared. Here, `Node` has a field of type `int` and another field of type `*Node`.

```
    type Node struct {
        val int
        next *Node
    }
```

There are, however, situations where it's useful to write functions or structs where the specific *type* of a parameter or field is left unspecified until it is used.

The case for generic types is easy to understand. In "Code Your Methods for nil Instances" on page 133, we looked at a binary tree for ints. If we want a binary tree for strings or float64s and we wanted type safety, there are a few options. The first possibility is writing a custom tree for each type, but having that much duplicated code is verbose and error-prone.

Before Go added generics, the only way to avoid duplicated code would be to modify our tree implementation so that it uses an interface to specify how to order values. The interface would look like this:

```
    type Orderable interface {
        // Order returns:
        // a value < 0 when the Orderable is less than the supplied value,
        // a value > 0 when the Orderable is greater than the supplied value,
        // and 0 when the two values are equal.
        Order(interface{}) int
    }
```

Now that we have `Orderable`, we can modify our `Tree` implementation to support it:

```
    type Tree struct {
        val         Orderable
        left, right *Tree
    }

    func (t *Tree) Insert(val Orderable) *Tree {
        if t == nil {
            return &Tree{val: val}
        }

        switch comp := val.Order(t.val); {
        case comp < 0:
            t.left = t.left.Insert(val)
        case comp > 0:
            t.right = t.right.Insert(val)
        }
        return t
    }
```

With an OrderableInt type, we can then insert int values:

```go
type OrderableInt int

func (oi OrderableInt) Order(val interface{}) int {
    return int(oi - val.(OrderableInt))
}

func main() {
    var it *Tree
    it = it.Insert(OrderableInt(5))
    it = it.Insert(OrderableInt(3))
    // etc...
}
```

While this code works correctly, it doesn't allow the compiler to validate that the values inserted into our data structure are all the same. If we also had an OrderableString type:

```go
type OrderableString string

func (os OrderableString) Order(val interface{}) int {
    return strings.Compare(string(os), val.(string))
}
```

The following code compiles:

```go
var it *Tree
it = it.Insert(OrderableInt(5))
it = it.Insert(OrderableString("nope"))
```

The Order function uses interface{} to represent the value that's passed in. This effectively short-circuits one of Go's primary advantages, compile-time type safety checking. When we compile code that attempts to insert an OrderableString into a Tree that already contains an OrderableInt, the compiler accepts the code. However, the program panics when run:

```
panic: interface conversion: interface {} is main.OrderableInt, not string
```

Now that Go has added generics, there's a way to implement a data structure once for multiple types and detect incompatible data at compile-time. We'll see how to properly use them in just a bit.

While data structures without generics are inconvenient, the real limitation is in writing functions. Several implementation decisions in Go's standard library were made because generics weren't originally part of the language. For example, rather than write multiple functions to handle different numeric types, Go implements functions like math.Max, math.Min, and math.Mod using float64 parameters, which have a range big enough to represent nearly every other numeric type exactly. (The exceptions are an int, int64, or uint with a value greater than $2^{53} - 1$ or less than $-2^{53} - 1$.)

There are other things that are impossible without generics. You cannot create a new instance of a variable that's specified by interface, nor can you specify that two parameters that are of the same interface type are also of the same concrete type. Without generics, you cannot write a function to process a slice of any type without resorting to reflection and giving up some performance along with compile-time type safety (this is how `sort.Slice` works). This meant that historically, functions that operate on slices would be repeated for each type of slice.

In 2017, I wrote a blog post called *Closures Are the Generics for Go* that explored using closures to work around some of these issues. However, the closure approach has several drawbacks. It is far less readable, forces values to escape to the heap, and simply doesn't work in many common situations.

The result is that many common algorithms, such as map, reduce, and filter, end up being reimplemented for different types. While simple algorithms are easy enough to copy, many (if not most) software engineers find it grating to duplicate code simply because the compiler isn't smart enough to do it automatically.

Introducing Generics in Go

Since the first announcement of Go, there have been calls for generics to be added to the language. Russ Cox, the development lead for Go, wrote a blog post in 2009 to explain why generics weren't initially included. Go emphasizes a fast compiler, readable code, and good execution time, and none of the generics implementations that they were aware of would allow them to include all three. After a decade studying the problem, the Go team has a workable approach, which is outlined in the Type Parameters Proposal.

We'll see how generics work in Go by looking at a stack. If you don't have a computer science background, a stack is a data type where values are added and removed in last in-first out (LIFO) order. It's like a pile of dishes waiting to be washed; the ones that were placed first are at the bottom, and you only get to them by working through the ones that were added later. Let's see how to make a stack using generics:

```go
type Stack[T any] struct {
    vals []T
}

func (s *Stack[T]) Push(val T) {
    s.vals = append(s.vals, val)
}

func (s *Stack[T]) Pop() (T, bool) {
    if len(s.vals) == 0 {
        var zero T
        return zero, false
    }
```

```
        top := s.vals[len(s.vals)-1]
        s.vals = s.vals[:len(s.vals)-1]
        return top, true
    }
```

There are a few things to note. First, we have `[T any]` after the type declaration. Type parameters are placed within brackets. They are written just like variable parameters, with the type name first and the type constraint second. You can pick any name for the type parameter, but it is customary to use capital letters for them. Go uses interfaces to specify which types can be used. If any type is usable, this is specified with the new universe block identifier `any`, which is exactly equivalent to `interface{}`. (starting with Go 1.18 and later, you can use `any` anywhere in your code where you would have used `interface{}`, but be aware that your code will not compile with versions of Go before 1.18.) Inside the `Stack` declaration, we declare `vals` to be of type `[]T`.

Next, we look at our method declarations. Just like we used `T` in our `vals` declaration, we do the same here. We also refer to the type in the receiver section with `Stack[T]` instead of `Stack`.

Finally, generics make zero value handling a little interesting. In `Pop`, we can't just return `nil`, because that's not a valid value for a value type, like `int`. The easiest way to get a zero value for a generic is to simply declare a variable with `var` and return it, since by definition, `var` always initializes its variable to the zero value if no other value is assigned.

Using a generic type is very similar to using a nongeneric one:

```
func main() {
    var intStack Stack[int]
    intStack.Push(10)
    intStack.Push(20)
    intStack.Push(30)
    v, ok := intStack.Pop()
    fmt.Println(v, ok)
}
```

The only difference is that when we declare our variable, we include the type that we want to use with our `Stack`, in this case `int`. If you try to push a string onto our stack, the compiler will catch it. Adding the line:

```
intStack.Push("nope")
```

produces the compiler error:

```
cannot use "nope" (untyped string constant) as int value
  in argument to intStack.Push
```

You can try out our generic stack on The Go Playground .

Let's add another method to our stack to tell us if the stack contains a value:

```
func (s Stack[T]) Contains(val T) bool {
    for _, v := range s.vals {
        if v == val {
            return true
        }
    }
    return false
}
```

Unfortunately, this does not compile. It gives the error:

```
invalid operation: v == val (type parameter T is not comparable with ==)
```

Just as `interface{}` doesn't say anything, neither does `any`. We can only store values of `any` type and retrieve them. To use `==`, we need a different type. Since nearly all Go types can be compared with `==` and `!=`, a new built-in interface called `comparable` is defined in the universe block. If we change our definition of `Stack` to use `comparable`:

```
type Stack[T comparable] struct {
    vals []T
}
```

we can then use our new method:

```
func main() {
    var s Stack[int]
    s.Push(10)
    s.Push(20)
    s.Push(30)
    fmt.Println(s.Contains(10))
    fmt.Println(s.Contains(5))
}
```

This prints out:

```
true
false
```

You can try out this updated stack as well.

Later on, we'll see how to make a generic binary tree. Before we do so, we're going to cover some additional concepts: *generic functions*, how generics work with interfaces, and *type terms*.

Generic Functions Abstract Algorithms

As we have hinted, you can also write generic functions. Earlier we mentioned that not having generics made it difficult to write map, reduce, and filter implementations that work for all types. Generics make it easy. Here are implementations from the type parameters proposal:

```go
// Map turns a []T1 to a []T2 using a mapping function.
// This function has two type parameters, T1 and T2.
// This works with slices of any type.
func Map[T1, T2 any](s []T1, f func(T1) T2) []T2 {
    r := make([]T2, len(s))
    for i, v := range s {
        r[i] = f(v)
    }
    return r
}

// Reduce reduces a []T1 to a single value using a reduction function.
func Reduce[T1, T2 any](s []T1, initializer T2, f func(T2, T1) T2) T2 {
    r := initializer
    for _, v := range s {
        r = f(r, v)
    }
    return r
}

// Filter filters values from a slice using a filter function.
// It returns a new slice with only the elements of s
// for which f returned true.
func Filter[T any](s []T, f func(T) bool) []T {
    var r []T
    for _, v := range s {
        if f(v) {
            r = append(r, v)
        }
    }
    return r
}
```

Functions place their type parameters after the function name and before the variable parameters. `Map` and `Reduce` have two type parameters, both of `any` type, while `Filter` has one. When we run the code:

```go
words := []string{"One", "Potato", "Two", "Potato"}
filtered := Filter(words, func(s string) bool {
    return s != "Potato"
})
fmt.Println(filtered)
lengths := Map(filtered, func(s string) int {
    return len(s)
})
fmt.Println(lengths)
sum := Reduce(lengths, 0, func(acc int, val int) int {
    return acc + val
})
fmt.Println(sum)
```

we get the output:

```
[One Two]
[3 3]
6
```

Try it for yourself.

Generics and Interfaces

You can use any interface as a type constraint, not just `any` and `comparable`. For example, say you wanted to make a type that holds any two values of the same type, as long as the type implements `fmt.Stringer`. Generics make it possible to enforce this at compile-time:

```
type Pair[T fmt.Stringer] struct {
    Val1 T
    Val2 T
}
```

You can also create interfaces that have type parameters. For example, here's an interface with a method that compares against a value of the specified type and returns a `float64`. It also embeds `fmt.Stringer`:

```
type Differ[T any] interface {
    fmt.Stringer
    Diff(T) float64
}
```

We'll use these two types to create a comparison function. The function takes in two `Pair` instances that have fields of type `Differ`, and returns the `Pair` with the closer values:

```
func FindCloser[T Differ[T]](pair1, pair2 Pair[T]) Pair[T] {
    d1 := pair1.Val1.Diff(pair1.Val2)
    d2 := pair2.Val1.Diff(pair2.Val2)
    if d1 < d2 {
        return pair1
    }
    return pair2
}
```

Note that `FindCloser` takes in `Pair` instances that have fields that meet the `Differ` interface. `Pair` requires that its fields are both of the same type and that the type meets the `fmt.Stringer` interface; this function is more selective. If the fields in a `Pair` instance don't meet `Differ`, the compiler will prevent you from using that `Pair` instance with `FindCloser`.

We now define a couple of types that meet the `Differ` interface:

```
type Point2D struct {
    X, Y int
}
```

```
func (p2 Point2D) String() string {
    return fmt.Sprintf("{%d,%d}", p2.X, p2.Y)
}

func (p2 Point2D) Diff(from Point2D) float64 {
    x := p2.X - from.X
    y := p2.Y - from.Y
    return math.Sqrt(float64(x*x) + float64(y*y))
}

type Point3D struct {
    X, Y, Z int
}

func (p3 Point3D) String() string {
    return fmt.Sprintf("{%d,%d,%d}", p3.X, p3.Y, p3.Z)
}

func (p3 Point3D) Diff(from Point3D) float64 {
    x := p3.X - from.X
    y := p3.Y - from.Y
    z := p3.Z - from.Z
    return math.Sqrt(float64(x*x) + float64(y*y) + float64(z*z))
}
```

And here's what it looks like to use this code:

```
func main() {
    pair2Da := Pair[Point2D]{Point2D{1, 1}, Point2D{5, 5}}
    pair2Db := Pair[Point2D]{Point2D{10, 10}, Point2D{15, 5}}
    closer := FindCloser(pair2Da, pair2Db)
    fmt.Println(closer)

    pair3Da := Pair[Point3D]{Point3D{1, 1, 10}, Point3D{5, 5, 0}}
    pair3Db := Pair[Point3D]{Point3D{10, 10, 10}, Point3D{11, 5, 0}}
    closer2 := FindCloser(pair3Da, pair3Db)
    fmt.Println(closer2)
}
```

Run it for yourself on The Go Playground.

Use Type Terms to Specify Operators

There's one more thing that we need to represent with generics: operators. If we want to write a generic version of `Min`, we need a way to specify that we can use comparison operators, like `<` and `>`. Go generics do that with a *type element*, which is composed of one or more *type terms* within an interface:

```
type BuiltInOrdered interface {
    string | int | int8 | int16 | int32 | int64 | float32 | float64 |
        uint | uint8 | uint16 | uint32 | uint64 | uintptr
}
```

We've already seen interfaces that have type elements in "Embedding and Interfaces" on page 146 . In that situation, we were embedding another interface to indicate that the method set of the containing interface includes the methods of the embedded interface. Here, we are listing concrete types separated by `|`. This specifies which types can be assigned to a type parameter and which operators are supported. The allowed operators are the ones that are valid for *all* of the listed types. In this case, those are the operators ==, !=, >, <, >=, <=, and +. Be aware that interfaces with concrete type terms in a type element are only valid as type parameter bounds. It is a compile-time error to use them as the type for a variable, field, return value, or parameter.

Now we can write our generic version of `Min` and use it with the built-in `int` type (or any of the other types listed in `BuiltInOrdered`):

```
func Min[T BuiltInOrdered](v1, v2 T) T {
    if v1 < v2 {
        return v1
    }
    return v2
}

func main() {
    a := 10
    b := 20
    fmt.Println(Min(a, b))
}
```

By default, type terms match exactly. If we try to use `Min` with a user-defined type whose underlying type is one of the types listed in `BuiltInOrdered`, we'll get an error. This code:

```
type MyInt int
var myA MyInt = 10
var myB MyInt = 20
fmt.Println(Min(myA, myB))
```

Produces the error:

```
MyInt does not implement BuiltInOrdered (possibly missing ~ for
int in constraint BuiltInOrdered)
```

The error text gives a hint for how to solve this problem. If you want a type term to be valid for any type that has the type term as its underlying type, put a ~ before the type term. This changes our definition of `BuiltInOrdered` to:

```
type BuiltInOrdered interface {
    ~string | ~int | ~int8 | ~int16 | ~int32 | ~int64 | ~float32 | ~float64 |
        ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~uintptr
}
```

You can look at this `Min` function on The Go Playground.

It is legal to have both type elements and method elements in an interface used for a type parameter. For example, you could specify that a type must have an underlying type of `int` and a `String() string` method:

```
type PrintableInt interface {
    ~int
    String() string
}
```

Be aware that Go will let you declare a type parameter interface that is impossible to actually instantiate. If we had used `int` instead of `~int` in `PrintableInt`, there would be no valid type that meets it, since `int` has no methods. This might seem bad, but the compiler still comes to your rescue. If you declare a type or function with an impossible type parameter, any attempt to use it causes a compiler error. Assume we declare these types:

```
type ImpossiblePrintableInt interface {
    int
    String() string
}

type ImpossibleStruct[T ImpossiblePrintableInt] struct {
    val T
}

type MyInt int

func (mi MyInt) String() string {
    return fmt.Sprint(mi)
}
```

Even though we cannot instantiate `ImpossibleStruct`, the compiler has no problem with any of these declarations. However, once we try using `ImpossibleStruct`, the compiler complains. This code:

```
s := ImpossibleStruct[int]{10}
s2 := ImpossibleStruct[MyInt]{10}
```

Produces the compile-time errors:

```
int does not implement ImpossiblePrintableInt (missing String method)
MyInt does not implement ImpossiblePrintableInt (possibly missing ~ for
int in constraint ImpossiblePrintableInt)
```

Try this out on The Go Playground.

In addition to built-in primitive types, type terms can also be slices, maps, arrays, channels, structs, or even functions. They are most useful when you want to ensure that a type parameter has a specific underlying type and one or more methods.

Type Inference and Generics

Just as Go supports type inference when using the `:=` operator, it also supports type inference to simplify calls to generic functions. You can see this in the calls to `Map`, `Filter`, and `Reduce` above. In some situations, type inference isn't possible (for example, when a type parameter is only used as a return value). When that happens, all type arguments must be specified. Here's a slightly silly bit of code that demonstrates a situation where type inference doesn't work:

```go
type Integer interface {
    int | int8 | int16 | int32 | int64 | uint | uint8 | uint16 | uint32 | uint64
}

func Convert[T1, T2 Integer](in T1) T2 {
    return T2(in)
}

func main() {
    var a int = 10
    b := Convert[int, int64](a) // can't infer the return type
    fmt.Println(b)
}
```

Try it out on The Go Playground.

Type Elements Limit Constants

Type elements also specify which constants can be assigned to variables of the generic type. Like operators, the constants need to be valid for all the type terms in the type element. There are no constants that can be assigned to every listed type in `BuiltInOrdered`, so you cannot assign a constant to a variable of that generic type. If you use the `Integer` interface, the following code will not compile, because you cannot assign 1,000 to an 8-bit integer:

```go
// INVALID!
func PlusOneThousand[T Integer](in T) T {
    return in + 1_000
}
```

However, this is valid:

```go
// VALID
func PlusOneHundred[T Integer](in T) T {
    return in + 100
}
```

Combining Generic Functions with Generic Data Structures

Let's return to our binary tree example and see how to combine everything we've learned to make a single tree that works for any concrete type.

The secret is to realize that what our tree needs is a single generic function that compares two values and tells us their order:

```go
type OrderableFunc [T any] func(t1, t2 T) int
```

Now that we have `OrderableFunc`, we can modify our tree implementation slightly. First, we're going to split it into two types, `Tree` and `Node` :

```go
type Tree[T any] struct {
    f    OrderableFunc[T]
    root *Node[T]
}

type Node[T any] struct {
    val         T
    left, right *Node[T]
}
```

We construct a new `Tree` with a constructor function:

```go
func NewTree[T any](f OrderableFunc[T]) *Tree[T] {
    return &Tree[T]{
        f: f,
    }
}
```

`Tree` 's methods are very simple, because they just call `Node` to do all the real work:

```go
func (t *Tree[T]) Add(v T) {
    t.root = t.root.Add(t.f, v)
}

func (t *Tree[T]) Contains(v T) bool {
    return t.root.Contains(t.f, v)
}
```

The `Add` and `Contains` methods on `Node` are very similar to what we've seen before. The only difference is that the function we are using to order our elements is passed in:

```go
func (n *Node[T]) Add(f OrderableFunc[T], v T) *Node[T] {
    if n == nil {
        return &Node[T]{val: v}
    }
    switch r := f(v, n.val); {
    case r <= -1:
        n.left = n.left.Add(f, v)
    case r >= 1:
        n.right = n.right.Add(f, v)
    }
    return n
}

func (n *Node[T]) Contains(f OrderableFunc[T], v T) bool {
    if n == nil {
        return false
    }
    switch r := f(v, n.val); {
    case r <= -1:
        return n.left.Contains(f, v)
    case r >= 1:
        return n.right.Contains(f, v)
    }
    return true
}
```

Now we need a function that matches the `OrderedFunc` definition. By taking advantage of `BuiltInOrdered`, we can write a single function that supports any primitive type:

```
func BuiltInOrderable[T BuiltInOrdered](t1, t2 T) int {
    if t1 < t2 {
        return -1
    }
    if t1 > t2 {
        return 1
    }
    return 0
}
```

When we use `BuiltInOrderable` with our `Tree`, it looks like this:

```
t1 := NewTree(BuiltInOrderable[int])
t1.Add(10)
t1.Add(30)
t1.Add(15)
fmt.Println(t1.Contains(15))
fmt.Println(t1.Contains(40))
```

For structs, we have two options. We can write a function:

```
type Person struct {
    Name string
    Age int
}

func OrderPeople(p1, p2 Person) int {
    out := strings.Compare(p1.Name, p2.Name)
    if out == 0 {
        out =  p1.Age - p2.Age
    }
    return out
}
```

Then we can pass that function in when we create our tree:

```
t2 := NewTree(OrderPeople)
t2.Add(Person{"Bob", 30})
t2.Add(Person{"Maria", 35})
t2.Add(Person{"Bob", 50})
fmt.Println(t2.Contains(Person{"Bob", 30}))
fmt.Println(t2.Contains(Person{"Fred", 25}))
```

Instead of using a function, we can also supply a method to `NewTree`. As we talked about in "Methods Are Functions Too" on page 134, you can use a method expression to treat a method like a function. Let's do that here. First we write the method:

```
func (p Person)Order(other Person) int {
    out := strings.Compare(p.Name, other.Name)
    if out == 0 {
        out =  p.Age - other.Age
    }
    return out
}
```

And then we use it:

```
t3 := NewTree(Person.Order)
t3.Add(Person{"Bob", 30})
t3.Add(Person{"Maria", 35})
t3.Add(Person{"Bob", 50})
fmt.Println(t3.Contains(Person{"Bob", 30}))
fmt.Println(t3.Contains(Person{"Fred", 25}))
```

You can find the code for this tree on The Go Playground.

Things That Are Left Out

Go remains a small, focused language, and the generics implementation for Go doesn't include many features that are found in generics implementations in other languages. Here are some of the features that are not in the initial implementation of Go generics.

While we can build a single tree that works with both user-defined and built-in types, languages like Python, Ruby, and C++ solve this problem in a different way. They include *operator overloading*, which allows user-defined types to specify implementations for operators. Go will not be adding this feature. This means that you can't use `range` to iterate over user-defined container types or `[]` to index into them.

There are good reasons for leaving out operator overloading. For one thing, there are a surprisingly large number of operators in Go. Go also doesn't have function or method overloading, and you'd need a way to specify different operator functionality for different types. Furthermore, operator overloading can lead to code that's harder to follow as developers invent clever meanings for symbols (in C++, << means "shift bits left" for some types and "write the value on the right to the value on the left" for others). These are the sorts of readability issues that Go tries to avoid.

Another useful feature that's been left out of the initial Go generics implementation is additional type parameters on methods. Looking back on the `Map/Reduce/Filter` functions, you might think they'd be useful as methods, like this:

```go
type functionalSlice[T any] []T

// THIS DOES NOT WORK
func (fs functionalSlice[T]) Map[E any](f func(T) E) functionalSlice[E] {
    out := make(functionalSlice[E], len(fs))
    for i, v := range fs {
        out[i] = f(v)
    }
    return out
}

// THIS DOES NOT WORK
func (fs functionalSlice[T]) Reduce[E any](start E, f func(E, T) E) E {
    out := start
    for _, v := range fs {
        out = f(out, v)
    }
    return out
}
```

which you could use like this:

```go
var numStrings = functionalSlice[string]{"1", "2", "3"}
sum := numStrings.Map(func(s string) int {
    v, _ := strconv.Atoi(s)
    return v
}).Reduce(0, func(acc int, cur int) int {
    return acc + cur
})
```

Unfortunately for fans of functional programming, this does not work. Rather than chaining method calls together, you need to either nest function calls or use the much more readable approach of invoking the functions one at a time and assigning the intermediate values to variables. The type parameter proposal goes into detail on the reasons for excluding parameterized methods.

There are also no variadic type parameters. In "Build Functions with Reflection to Automate Repetitive Tasks" on page 314, we wrote a wrapper function using reflection to time any existing function. Those must still be handled via reflection as there's no way to do this with generics. Any time you use type parameters, you must explicitly provide a name for each type you need, so you cannot represent a function with any number of parameters of different types.

Other features left out of Go generics are more esoteric. These include:

Specialization
: A function or method can be overloaded with one or more type-specific versions in addition to the generic version. Since Go doesn't have overloading, this feature is not under consideration.

Currying
: Allows you to partially instantiate a function or type based on another generic function or type by specifying some of the type parameters.

Metaprogramming
: Allows you to specify code that runs at compile-time to produce code that runs at runtime.

Idiomatic Go and Generics

Adding generics clearly changes some of the advice for how to use Go idiomatically. The use of `float64` to represent any numeric type will end. You should use `any` instead of `interface{}` to represent an unspecified type in a data structure or function parameter. You can handle different slice types with a single function. But don't feel the need to switch all of your code over to using type parameters immediately. Your old code will still work as new design patterns are invented and refined.

It's still too early to judge the long-term impact of generics on performance. The compiler in Go 1.18 is slower than in previous versions, but this is expected to be addressed in future releases. There has already been some research on the current runtime impact. Vicent Marti wrote a detailed blog post where he explores cases where generics result in slower code and the implementation details that explain why this is so. Conversely, Eli Bendersky wrote a blog post that shows that generics make sorting algorithms faster. Again, as the generics implementation matures in future versions of Go, expect runtime performance to improve.

As always, the goal is to write maintainable programs that are fast enough to meet your needs. Use the benchmarking and profiling tools we discussed in "Benchmarks" on page 285 to measure and improve.

Further Futures Unlocked

The initial release of generics in Go 1.18 was very conservative. It added the new interfaces `any` and `comparable` to the universe block, but there were no API changes in the standard library to support generics. A stylistic change has been made; virtually all uses of `interface{}` in the standard library were replaced with `any`.

It is likely that future versions of the standard library will include new interface definitions to represent common cases (like `Orderable`), new types (like a set, tree, or ordered map), and new functions. Feel free to write your own in the meantime, but consider replacing them once the standard library is updated.

Generics might be the basis for other future features. One possibility is *sum types*. Just as type elements are used to specify the types that can be substituted for a type parameter, they could also be used for interfaces in variable parameters. This would enable some interesting features. Today, Go has a problem with a common situation in JSON: a field that can be a single value or a list of values. Even with generics, the only way to handle this is with a field of type `any`. Adding sum types would allow you to create an interface that specifies that a field could be a string, a slice of strings, and nothing else. A type switch could then completely enumerate every valid type, improving type safety. This ability to specify a bounded set of types allows many modern languages (including Rust and Swift) to use sum types to represent enums. Given the weakness of Go's current enum features, this might be an attractive solution, but it will take time for these ideas to be evaluated and explored.

Wrapping Up

In this chapter, we took a look at generics and how to use them to simplify our code. It's still early days for generics in Go. It will be exciting to see how they help grow the language while still maintaining the spirit that make Go special.

We've completed our journey through Go and how to use it idiomatically. Like any graduation ceremony, it's time for a few closing words. Let's look back at what was said in the preface. "[P]roperly written, Go is boring….Well-written Go programs tend to be straightforward and sometimes a bit repetitive." I hope you can now see why this leads to better software engineering. Idiomatic Go is a set of tools, practices, and patterns that makes it easier to maintain software across time and changing teams. That's not to say the cultures around other languages don't value maintainability; it just may not be their highest priority. Instead, they emphasize things like performance, new features, or concise syntax. There is a place for these trade-offs, but in the long run, I suspect Go's focus on crafting software that lasts will win out.

I wish you the best as you create the software for the next 50 years of computing.