CHAPTER 8
Generics

"Don't repeat yourself" is common software engineering advice. It's better to reuse a data structure or a function than it is to re-create it, because it's hard to keep code changes in sync between duplicated code. In a strongly typed language like Go, the type of every function parameter and every struct field must be known at compile time. This strictness enables the compiler to help validate that your code is correct, but sometimes when you'll want to reuse the logic in a function or the fields in a struct with different types. Go provides this functionality via type parameters, which are colloquially referred to as *generics*. In this chapter you'll learn why people want generics, what Go's implementation of generics can do, what generics can't do, and how to use them properly.

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 accustomed to 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 to think of them as "type parameters." So far in this book, you've seen functions that take in parameters whose values are specified when the function is called. In "Multiple Return Values" on page 96, the function `divAndRemainder` has two `int` parameters and returns two `int` values:

```
func divAndRemainder(num, denom int) (int, int, error) {
    if denom == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
    return num / denom, num % denom, nil
}
```

Similarly, you create structs by specifying the type for the fields 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
}
```

In some situations, however, it's useful to write functions or structs that leave the specific *type* of a parameter or field unspecified until it is used.

The case for generic types is easy to understand. In "Code Your Methods for nil Instances" on page 148, you looked at a binary tree for `int`s. If you want a binary tree for strings or float64s and want type safety, you have 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.

Without generics, the only way to avoid duplicated code would be to modify your 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(any) int
}
```

Now that you have `Orderable`, you can modify your `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, you can then insert `int` values:

```
type OrderableInt int

func (oi OrderableInt) Order(val any) 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 your data structure are all the same. If you also had an `OrderableString` type:

```
type OrderableString string

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

the following code compiles:

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

The `Order` function uses `any` to represent the value that's passed in. This effectively short-circuits one of Go's primary advantages: checking compile-time type safety. When you 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
```

You can try out this code in the *sample_code/non_generic_tree* directory in the Chapter 8 repository.

With generics, there's a way to implement a data structure once for multiple types and detect incompatible data at compile time. You'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 $-2^{53} - 1$.)

Some other things are impossible without generics. You cannot create a new instance of a variable that's specified by an interface, nor can you specify that two parameters 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 before generics were introduced to Go, functions that operate on slices (like `map`, `reduce`, and `filter`) would be repeatedly implemented for each type of slice. 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, people have called 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.

You can 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 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 get to them only by working through the ones that were added later. Let's see how to make a stack using generics:

```
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, you have `[T any]` after the type declaration. Type parameter information is placed within brackets and has two parts. The first part is the type parameter name. You can pick any name for the type parameter, but using capital letters is customary. The second part is the *type constraint*, which uses a Go interface to specify which types are valid. If any type is usable, this is specified with the universe block identifier `any`, which you first saw in "The Empty Interface Says Nothing" on page 166. Inside the `Stack` declaration, you declare `vals` to be of type `[]T`.

Next, look at the method declarations. Just as you used `T` in your `vals` declaration, you do the same here. You 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`, you 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 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 you declare your variable, you include the type that you want to use with your `Stack`—in this case, `int`. If you try to push a string onto your 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 the generic stack on The Go Playground or in the *sample_code/stack* directory in the Chapter 8 repository.

Add another method to your stack to tell you 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 an error:

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

Just as `interface{}` doesn't say anything, neither does `any`. You can only store values of `any` type and retrieve them. To use `==`, you 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 you change the definition of `Stack` to use `comparable`:

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

you can then use your 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 the following:

```
true
false
```

You can try out this updated stack on The Go Playground or in the *sample_code/comparable_stack* directory in the Chapter 8 repository.

Later, you'll see how to make a generic binary tree. First, I'll cover some additional concepts: *generic functions*, how generics work with interfaces, and *type terms*.

Generic Functions Abstract Algorithms

As I have hinted, you can also write generic functions. Earlier I 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:

```
// 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 you run the code:

```
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)
```

you get the output:

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

Try it for yourself on The Go Playground or in the *sample_code/map_filter_reduce* directory in the Chapter 8 repository.

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
}
```

You'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`.

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 or in the *sample_code/generic_interface* directory in the Chapter 8 repository.

Use Type Terms to Specify Operators

One more thing needs to be represented with generics: operators. The `divAndRemainder` function works fine with `int`, but using it with other integer types requires type casting, and `uint` allows you to represent values that are too big for an `int`. If you want to write a generic version of `divAndRemainder`, you need a way to specify that you can use `/` and `%`. Go generics do that with a *type element*, which is composed of one or more *type terms* within an interface:

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

In "Embedding and Interfaces" on page 162, you learned about embedding interfaces to indicate that the method set of the containing interface includes the methods of the embedded interface. Type elements specify which types can be assigned to a type parameter and which operators are supported. They list concrete types separated by `|`. The allowed operators are the ones that are valid for *all* of the listed types. The modulus (`%`) operator is valid only for integers, so we list all integer types. (You can leave off `byte` and `rune` because they are type aliases for `uint8` and `int32`, respectively.)

Be aware that interfaces with type elements are valid only as type constraints. It is a compile-time error to use them as the type for a variable, field, return value, or parameter.

Now you can write your generic version of `divAndRemainder` and use it with the built-in `uint` type (or any of the other types listed in `Integer`):

```
func divAndRemainder[T Integer](num, denom T) (T, T, error) {
    if denom == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
    return num / denom, num % denom, nil
}

func main() {
    var a uint = 18_446_744_073_709_551_615
    var b uint = 9_223_372_036_854_775_808
    fmt.Println(divAndRemainder(a, b))
}
```

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

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

produces the following error:

```
MyInt does not satisfy Integer (possibly missing ~ for int in Integer)
```

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 the definition of `Integer` as follows:

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

You can look at the generic version of the `divAndRemainder` function on The Go Playground or in the *sample_code/type_terms* directory in the Chapter 8 repository.

The addition of type terms allows you to define a type that lets you write generic comparison functions:

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

The `Ordered` interface lists all types that support the ==, `!=`, <, >, <=, and >= operators. Having a way to specify that a variable represents an orderable type is so useful that Go 1.21 added the `cmp` package, which defines this `Ordered` interface. The package also defines two comparison functions. The `Compare` function returns either –1, 0, or 1, depending on whether its first parameter is less than, equal to, or greater than its second parameter, and the `Less` function returns `true` if its first parameter is less than its second parameter.

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

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

produces the following 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 on The Go Playground or in the *sample_code/impossible* directory in the Chapter 8 repository.

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` earlier. In some situations, type inference isn't possible (for example, when a type parameter is used only 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:

```
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 or in the *sample_code/type_inference* directory in the Chapter 8 repository.

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 `Ordered`, 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 the value 1,000 to an 8-bit integer:

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

However, this is valid:

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

Combining Generic Functions with Generic Data Structures

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

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

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

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

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

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

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

```
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:

```
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 you've seen before. The only difference is that the function you are using to order your elements is passed in:

```
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 you need a function that matches the `OrderedFunc` definition. Luckily, you've already seen one: `Compare` in the `cmp` package. When you use it with your `Tree`, it looks like this:

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

For structs, you have two options. You can write a function:

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

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

Then you can pass that function in when you create your 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, you can also supply a method to `NewTree`. As I talked about in "Methods Are Functions Too" on page 149, you can use a method expression to treat a method like a function. Let's do that here. First, write the method:

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

And then 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 or in the *sample_code/generic_tree* directory in the Chapter 8 repository.

More on comparable

As you saw in "Interfaces Are Comparable" on page 165, interfacesare one of the comparable types in Go. This means that you need to be careful when using `==` and `!=` with variables of the interface type. If the underlying type of the interface is not comparable, your code panics at runtime.

This pothole still exists when using the `comparable` interface with generics. Say you've defined an interface and a couple of implementations:

```
type Thinger interface {
    Thing()
}

type ThingerInt int

func (t ThingerInt) Thing() {
    fmt.Println("ThingInt:", t)
}

type ThingerSlice []int

func (t ThingerSlice) Thing() {
    fmt.Println("ThingSlice:", t)
}
```

You also define a generic function that accepts only values that are `comparable`:

```
func Comparer[T comparable](t1, t2 T) {
    if t1 == t2 {
        fmt.Println("equal!")
    }
}
```

It's legal to call this function with variables of type `int` or `ThingerInt`:

```
var a int = 10
var b int = 10
Comparer(a, b) // prints true

var a2 ThingerInt = 20
var b2 ThingerInt = 20
Comparer(a2, b2) // prints true
```

The compiler won't allow you to call this function with variables of type `ThingerSlice` (or `[]int`):

```
var a3 ThingerSlice = []int{1, 2, 3}
var b3 ThingerSlice = []int{1, 2, 3}
Comparer(a3, b3) // compile fails: "ThingerSlice does not satisfy comparable"
```

However, it's perfectly legal to call it with variables of type `Thinger`. If you use `ThingerInt`, the code compiles and works as expected:

```
var a4 Thinger = a2
var b4 Thinger = b2
Comparer(a4, b4) // prints true
```

But you can also assign `ThingerSlice` to variables of type `Thinger`. That's where things go wrong:

```
a4 = a3
b4 = b3
Comparer(a4, b4) // compiles, panics at runtime
```

The compiler won't stop you from building this code, but if you run it, your program will panic with the message `panic: runtime error: comparing uncomparable type main.ThingerSlice` (see "panic and recover" on page 218 for more information). You can try this code yourself on The Go Playground or in the *sample_code/more_comparable* directory in the Chapter 8 repository.

For more technical details on how comparable types and generics interact and why this design decision was made, read the blog post "All Your Comparable Types" from Robert Griesemer on the Go team.

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. This section describes some of the features that are not in the initial implementation of Go generics.

While you 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, Go has a surprisingly large number of operators. 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:

```
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:

```
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.

Go also has no variadic type parameters. As discussed in "Variadic Input Parameters and Slices" on page 95, to implement a function that takes in a varying number of parameters, you specify that the last parameter type starts with `...`. For example, there's no way to specify some sort of type pattern to those variadic parameters, such as alternating `string` and `int`. All variadic variables must match a single declared type, which can be generic or not.

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

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 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. As of this writing, there's no impact on compilation time. The Go 1.18 compiler was slower than previous versions, but the compiler in Go 1.20 resolved this issue.

Some research has also been done on the runtime impact of generics. Vicent Marti wrote a detailed blog post exploring 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.

In particular, do not change a function that has an interface parameter into a function with a generic type parameter in hopes of improving performance. For example, converting this trivial function:

```
type Ager interface {
    age() int
}

func doubleAge(a Ager) int {
    return a.age() * 2
}
```

into:

```
func doubleAgeGeneric[T Ager](a T) int {
    return a.age() * 2
}
```

makes the function call about 30% slower in Go 1.20. (For a nontrivial function, there's no significant performance difference.) You can run the benchmarks using the code in the *sample_code/perf* directory in the Chapter 8 repository.

This might be surprising for developers experienced with generics in other languages. In C++, for example, the compiler uses generics on abstract data types to turn what is normally a runtime operation (figuring out which concrete type is being used) into a compile-time one, generating a unique function for each concrete type. This makes the resulting binary larger but also faster. As Vicent explains in his blog post, the current Go compiler generates only unique functions for different underlying types. Furthermore, all pointer types share a single generated function. To differentiate between the different types that are passed in to shared generated functions, the compiler adds additional runtime lookups. This causes the slowdown in performance.

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 that are discussed in "Using Benchmarks" on page 393 to measure and improve your code.

Adding Generics to the Standard Library

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 no API changes occurred 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`.

Now that the Go community is more comfortable with generics, we are starting to see more changes. Starting with Go 1.21, the standard library includes functions that use generics to implement common algorithms for slices, maps, and concurrency. In Chapter 3, I covered the `Equal` and `EqualFunc` functions in the `slices` and `maps` packages. Other functions in these packages simplify slice and map manipulation. The `Insert`, `Delete`, and `DeleteFunc` functions in the `slices` package allow developers to avoid building some surprisingly tricky slice-handling code. The `maps.Clone` function takes advantage of the Go runtime to provide a faster way to create a shallow copy of a map. In "Run Code Exactly Once" on page 308, you'll learn about `sync.OnceValue` and `sync.OnceValues`, which use generics to build functions that are invoked only once and return one or two values. Prefer using functions in these packages over writing your own implementations. Future versions of the standard library will likely include additional new functions and types that take advantage of generics.

Future Features Unlocked

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 specifying 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.

Exercises

Now that you've seen how generics work, apply them to solve the following problems. Solutions are available in the *exercise_solutions* directory in the Chapter 8 repository.

1. Write a generic function that doubles the value of any integer or float that's passed in to it. Define any needed generic interfaces.
2. Define a generic interface called `Printable` that matches a type that implements `fmt.Stringer` and has an underlying type of `int` or `float64`. Define types that meet this interface. Write a function that takes in a `Printable` and prints its value to the screen using `fmt.Println`.
3. Write a generic singly linked list data type. Each element can hold a comparable value and has a pointer to the next element in the list. The methods to implement are as follows:

```
// adds a new element to the end of the linked list
Add(T)
// adds an element at the specified position in the linked list
Insert(T, int)
// returns the position of the supplied value, -1 if it's not present
Index (T) int
```

Wrapping Up

In this chapter, you took a look at generics and how to use them to simplify your 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 makes Go special.

In the next chapter, you are going to learn how to properly use one of Go's most controversial features: errors.