Generics

You can find all the code for this chapter here

This chapter will give you an introduction to generics, dispel reservations you may have about them and, give you an idea how to simplify some of your code in the future. After reading this you'll know how to write:

- A function that takes generic arguments
- A generic data-structure

Our own test helpers (`AssertEqual`, `AssertNotEqual`)

To explore generics we'll write some test helpers.

Assert on integers

Let's start with something basic and iterate toward our goal

```go
import "testing"

func TestAssertFunctions(t *testing.T) {
	t.Run("asserting on integers", func(t *testing.T) {
		AssertEqual(t, 1, 1)
		AssertNotEqual(t, 1, 2)
	})
}

func AssertEqual(t *testing.T, got, want int) {
    t.Helper()
	if got != want {
		t.Errorf("got %d, want %d", got, want)
	}
}

func AssertNotEqual(t *testing.T, got, want int) {
    t.Helper()
        if got == want {

        t.Errorf("didn't want %d", got)
        }
}
```

Assert on strings

Being able to assert on the equality of integers is great but what if we want to assert on `string` ?

```
t.Run("asserting on strings", func(t *testing.T) {
    AssertEqual(t, "hello", "hello")
    AssertNotEqual(t, "hello", "Grace")
})
```

You'll get an error

```
# github.com/quii/learn-go-with-tests/generics [github.com/quii/learn-go-with-tests/g
./generics_test.go:12:18: cannot use "hello" (untyped string constant) as int value i
./generics_test.go:13:21: cannot use "hello" (untyped string constant) as int value i
./generics_test.go:13:30: cannot use "Grace" (untyped string constant) as int value i
```

If you take your time to read the error, you'll see the compiler is complaining that we're trying to pass a `string` to a function that expects an `integer`.

Recap on type-safety

If you've read the previous chapters of this book, or have experience with statically typed languages, this should not surprise you. The Go compiler expects you to write your functions, structs e.t.c. by describing what types you wish to work with.

You can't pass a `string` to a function that expects an `integer`.

Whilst this can feel like ceremony, it can be extremely helpful. By describing these constraints you,

- Make function implementation simpler. By describing to the compiler what types you work with, you **constrain the number of possible valid implementations**. You can't "add" a `Person` and a `BankAccount`. You can't capitalise an `integer`. In software, constraints are often extremely helpful.

- Are prevented from accidentally passing data to a function you didn't mean to.

Go offers you a way to be more abstract with your types with interfaces, so that you can design functions that do not take concrete types but instead, types that offer the behaviour you need. This gives you some flexibility whilst maintaining type-safety.

A function that takes a string or an integer? (or indeed, other things)

Another option Go has to make your functions more flexible is by declaring the type of your argument as `interface{}` which means "anything".

Try changing the signatures to use this type instead.

```go
func AssertEqual(got, want interface{})

func AssertNotEqual(got, want interface{})
```

The tests should now compile and pass. If you try making them fail you'll see the output is a bit ropey because we're using the integer `%d` format string to print our messages, so change them to the general `%+v` format for a better output of any kind of value.

The problem with `interface{}`

Our `AssertX` functions are quite naive but conceptually aren't too different to how other popular libraries offer this functionality

```go
func (is *I) Equal(a, b interface{})
```

So what's the problem?

By using `interface{}` the compiler can't help us when writing our code, because we're not telling it anything useful about the types of things passed to the function. Try comparing two different types.

```go
AssertNotEqual(1, "1")
```

In this case, we get away with it; the test compiles, and it fails as we'd hope, although the error message `got 1, want 1` is unclear; but do we want to be able to compare strings with integers? What about comparing a `Person` with an `Airport`?

Writing functions that take `interface{}` can be extremely challenging and bug-prone because we've *lost* our constraints, and we have no information at compile time as to what kinds of data we're dealing with.

This means **the compiler can't help us** and we're instead more likely to have **runtime errors** which could affect our users, cause outages, or worse.

Often developers have to use reflection to implement these *ahem* generic functions, which can get complicated to read and write, and can hurt the performance of your program.

Our own test helpers with generics

Ideally, we don't want to have to make specific `AssertX` functions for every type we ever deal with. We'd like to be able to have *one* `AssertEqual` function that works with *any* type but does not let you compare apples and oranges.

Generics offer us a way to make abstractions (like interfaces) by letting us **describe our constraints**. They allow us to write functions that have a similar level of flexibility that `interface{}` offers but retain type-safety and provide a better developer experience for callers.

```go
func TestAssertFunctions(t *testing.T) {
	t.Run("asserting on integers", func(t *testing.T) {
		AssertEqual(t, 1, 1)
		AssertNotEqual(t, 1, 2)
	})

	t.Run("asserting on strings", func(t *testing.T) {
		AssertEqual(t, "hello", "hello")
		AssertNotEqual(t, "hello", "Grace")
	})

	// AssertEqual(t, 1, "1") // uncomment to see the error
}

func AssertEqual[T comparable](t *testing.T, got, want T) {
    t.Helper()
	if got != want {
```

```
                t.Errorf("got %v, want %v", got, want)
        }
}

func AssertNotEqual[T comparable](t *testing.T, got, want T) {
    t.Helper()
        if got == want {
                t.Errorf("didn't want %v", got)
        }
}
```

To write generic functions in Go, you need to provide "type parameters" which is just a fancy way of saying "describe your generic type and give it a label".

In our case the type of our type parameter is `comparable` and we've given it the label of `T`. This label then lets us describe the types for the arguments to our function (`got, want T`).

We're using `comparable` because we want to describe to the compiler that we wish to use the `==` and `!=` operators on things of type `T` in our function, we want to compare! If you try changing the type to `any`,

```
func AssertNotEqual[T any](got, want T)
```

You'll get the following error:

```
prog.go2:15:5: cannot compare got != want (operator != not defined for T)
```

Which makes a lot of sense, because you can't use those operators on every (or `any`) type.

Is a generic function with `T any` the same as `interface{}` ?

Consider two functions

```
func GenericFoo[T any](x, y T)
```

```
func InterfaceyFoo(x, y interface{})
```

What's the point of generics here? Doesn't `any` describe... anything?

In terms of constraints, `any` does mean "anything" and so does `interface{}`. In fact, `any` was added in 1.18 and is *just an alias for* `interface`.

The difference with the generic version is *you're still describing a specific type* and what that means is we've still constrained this function to only work with *one* type.

What this means is you can call `InterfaceyFoo` with any combination of types (e.g `InterfaceyFoo(apple, orange)`). However `GenericFoo` still offers some constraints because we've said that it only works with *one* type, `T`.

Valid:

- `GenericFoo(apple1, apple2)`
- `GenericFoo(orange1, orange2)`
- `GenericFoo(1, 2)`
- `GenericFoo("one", "two")`

Not valid (fails compilation):

- `GenericFoo(apple1, orange1)`
- `GenericFoo("1", 1)`

If your function returns the generic type, the caller can also use the type as it was, rather than having to make a type assertion because when a function returns `interface{}` the compiler cannot make any guarantees about the type.

Next: Generic data types

We're going to create a stack data type. Stacks should be fairly straightforward to understand from a requirements point of view. They're a collection of items where you can `Push` items to the "top" and to get items back again you `Pop` items from the top (LIFO - last in, first out).

For the sake of brevity I've omitted the TDD process that arrived me at the following code for a stack of `int`s, and a stack of `string`s.

```go
type StackOfInts struct {
	values []int
}

func (s *StackOfInts) Push(value int) {
	s.values = append(s.values, value)
}

func (s *StackOfInts) IsEmpty() bool {
	return len(s.values) == 0
}

func (s *StackOfInts) Pop() (int, bool) {
	if s.IsEmpty() {
		return 0, false
	}

	index := len(s.values) - 1
	el := s.values[index]
	s.values = s.values[:index]
	return el, true
}

type StackOfStrings struct {
	values []string
}

func (s *StackOfStrings) Push(value string) {
	s.values = append(s.values, value)
}

func (s *StackOfStrings) IsEmpty() bool {
	return len(s.values) == 0
}

func (s *StackOfStrings) Pop() (string, bool) {
	if s.IsEmpty() {
		return "", false
	}

	index := len(s.values) - 1
	el := s.values[index]
	s.values = s.values[:index]
	return el, true
}
```

I've created a couple of other assertion functions to help out

```go
func AssertTrue(t *testing.T, got bool) {
	t.Helper()
	if !got {
		t.Errorf("got %v, want true", got)
	}
}

func AssertFalse(t *testing.T, got bool) {
	t.Helper()
	if got {
		t.Errorf("got %v, want false", got)
	}
}
```

And here's the tests

```go
func TestStack(t *testing.T) {
	t.Run("integer stack", func(t *testing.T) {
		myStackOfInts := new(StackOfInts)

		// check stack is empty
		AssertTrue(t, myStackOfInts.IsEmpty())

		// add a thing, then check it's not empty
		myStackOfInts.Push(123)
		AssertFalse(t, myStackOfInts.IsEmpty())

		// add another thing, pop it back again
		myStackOfInts.Push(456)
		value, _ := myStackOfInts.Pop()
		AssertEqual(t, value, 456)
		value, _ = myStackOfInts.Pop()
		AssertEqual(t, value, 123)
		AssertTrue(t, myStackOfInts.IsEmpty())
	})

	t.Run("string stack", func(t *testing.T) {
		mySackOfStrings := new(StackOfStrings)

		// check stack is empty
		AssertTrue(t, myStackOfStrings.IsEmpty())

		// add a thing, then check it's not empty
		myStackOfStrings.Push("123")
		AssertFalse(t, myStackOfStrings.IsEmpty())

		// add another thing, pop it back again
		myStackOfStrings.Push("456")
		value, _ := myStackOfStrings.Pop()
		AssertEqual(t, value, "456")
		value, _ = myStackOfStrings.Pop()
		AssertEqual(t, value, "123")
		AssertTrue(t, myStackOfStrings.IsEmpty())
	})
}
```

Problems

- The code for both `StackOfStrings` and `StackOfInts` is almost identical. Whilst duplication isn't always the end of the world, it's more code to read, write and maintain.
- As we're duplicating the logic across two types, we've had to duplicate the tests too.

We really want to capture the *idea* of a stack in one type, and have one set of tests for them. We should be wearing our refactoring hat right now which means we should not be changing the tests because we want to maintain the same behaviour.

Without generics, this is what we *could* do

```go
type StackOfInts = Stack
type StackOfStrings = Stack

type Stack struct {
	values []interface{}
}

func (s *Stack) Push(value interface{}) {
	s.values = append(s.values, value)
}
```

```go
func (s *Stack) IsEmpty() bool {

        return len(s.values) == 0
}

func (s *Stack) Pop() (interface{}, bool) {
        if s.IsEmpty() {
                var zero interface{}
                return zero, false
        }

        index := len(s.values) - 1
        el := s.values[index]
        s.values = s.values[:index]
        return el, true
}
```

- We're aliasing our previous implementations of `StackOfInts` and `StackOfStrings` to a new unified type `Stack`
- We've removed the type safety from the `Stack` by making it so `values` is a slice of `interface{}`

To try this code, you'll have to remove the type constraints from our assert functions:

```go
func AssertEqual(t *testing.T, got, want interface{})
```

If you do this, our tests still pass. Who needs generics?

The problem with throwing out type safety

The first problem is the same as we saw with our `AssertEquals` - we've lost type safety. I can now `Push` apples onto a stack of oranges.

Even if we have the discipline not to do this, the code is still unpleasant to work with because when methods **return `interface{}` they are horrible to work with**.

Add the following test,

```go
t.Run("interface stack dx is horrid", func(t *testing.T) {
    myStackOfInts := new(StackOfInts)

    myStackOfInts.Push(1)
    myStackOfInts.Push(2)
    firstNum, _ := myStackOfInts.Pop()
    secondNum, _ := myStackOfInts.Pop()
    AssertEqual(firstNum+secondNum, 3)
})
```

You get a compiler error, showing the weakness of losing type-safety:

```
invalid operation: operator + not defined on firstNum (variable of type interface{})
```

When `Pop` returns `interface{}` it means the compiler has no information about what the data is and therefore severely limits what we can do. It can't know that it should be an integer, so it does not let us use the `+` operator.

To get around this, the caller has to do a type assertion for each value.

```go
t.Run("interface stack dx is horrid", func(t *testing.T) {
    myStackOfInts := new(StackOfInts)

    myStackOfInts.Push(1)
    myStackOfInts.Push(2)
    firstNum, _ := myStackOfInts.Pop()
    secondNum, _ := myStackOfInts.Pop()

    // get our ints from out interface{}
    reallyFirstNum, ok := firstNum.(int)
    AssertTrue(t, ok) // need to check we definitely got an int out of the interface{

    reallySecondNum, ok := secondNum.(int)
    AssertTrue(t, ok) // and again!

    AssertEqual(t, reallyFirstNum+reallySecondNum, 3)
})
```

The unpleasantness radiating from this test would be repeated for every potential user of our `Stack` implementation, yuck.

Generic data structures to the rescue

Just like you can define generic arguments to functions, you can define generic data structures.

Here's our new `Stack` implementation, featuring a generic data type.

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

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

func (s *Stack[T]) IsEmpty() bool {
    return len(s.values)==0
}

func (s *Stack[T]) Pop() (T, bool) {
    if s.IsEmpty() {
        var zero T
        return zero, false
    }

    index := len(s.values) -1
    el := s.values[index]
    s.values = s.values[:index]
    return el, true
}
```

Here's the tests, showing them working how we'd like them to work, with full type-safety.

```go
func TestStack(t *testing.T) {
	t.Run("integer stack", func(t *testing.T) {
		myStackOfInts := new(Stack[int])

		// check stack is empty
		AssertTrue(t, myStackOfInts.IsEmpty())

		// add a thing, then check it's not empty
		myStackOfInts.Push(123)
		AssertFalse(t, myStackOfInts.IsEmpty())
		// add another thing, pop it back again

		myStackOfInts.Push(456)
		value, _ := myStackOfInts.Pop()
		AssertEqual(t, value, 456)
		value, _ = myStackOfInts.Pop()
		AssertEqual(t, value, 123)
		AssertTrue(t, myStackOfInts.IsEmpty())

		// can get the numbers we put in as numbers, not untyped interface{}
		myStackOfInts.Push(1)
		myStackOfInts.Push(2)
		firstNum, _ := myStackOfInts.Pop()
		secondNum, _ := myStackOfInts.Pop()
		AssertEqual(t, firstNum+secondNum, 3)
	})
}
```

You'll notice the syntax for defining generic data structures is consistent with defining generic arguments to functions.

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

It's *almost* the same as before, it's just that what we're saying is the **type of the stack constrains what type of values you can work with**.

Once you create a `Stack[Orange]` or a `Stack[Apple]` the methods defined on our stack will only let you pass in and will only return the particular type of the stack you're working with:

```go
func (s *Stack[T]) Pop() (T, bool)
```

You can imagine the types of implementation being somehow generated for you, depending on what type of stack you create:

```go
func (s *Stack[Orange]) Pop() (Orange, bool)
func (s *Stack[Apple]) Pop() (Apple, bool)
```

Now that we have done this refactoring, we can safely remove the string stack test because we don't need to prove the same logic over and over.

Using a generic data type we have:

- Reduced duplication of important logic.
- Made `Pop` return `T` so that if we create a `Stack[int]` we in practice get back `int` from `Pop`; we can now use `+` without the need for type assertion gymnastics.
- Prevented misuse at compile time. You cannot `Push` oranges to an apple stack.

Wrapping up

This chapter should have given you a taste of generics syntax, and some ideas as to why generics might be helpful. We've written our own `Assert` functions which we can safely re-use to experiment with other ideas around generics, and we've implemented a simple data structure to store any type of data we wish, in a type-safe manner.

Generics are simpler than using `interface{}` in most cases

If you're inexperienced with statically-typed languages, the point of generics may not be immediately obvious, but I hope the examples in this chapter have illustrated where the Go language isn't as expressive as we'd like. In particular using `interface{}` makes your code:

- Less safe (mix apples and oranges), requires more error handling
- Less expressive, `interface{}` tells you nothing about the data
- More likely to rely on reflection, type-assertions etc which makes your code more difficult to work with and more error prone as it pushes checks from compile-time to runtime

Using statically typed languages is an act of describing constraints. If you do it well, you create code that is not only safe and simple to use but also simpler to write because the possible solution space is smaller.

Generics gives us a new way to express constraints in our code, which as demonstrated will allow us to consolidate and simplify code that is not possible to do today.

Will generics turn Go into Java?

- No.

There's a lot of FUD (fear, uncertainty and doubt) in the Go community about generics leading to nightmare abstractions and baffling code bases. This is usually caveatted with "they must be used carefully".

Whilst this is true, it's not especially useful advice because this is true of any language feature.

Not many people complain about our ability to define interfaces which, like generics is a way of describing constraints within our code. When you describe an interface you are making a design choice that *could be poor*, generics are not unique in their ability to make confusing, annoying to use code.

You're already using generics

When you consider that if you've used arrays, slices or maps; you've *already been a consumer of generic code*.

```
var myApples []Apples
// You cant do this!
append(myApples, Orange{})
```

Abstraction is not a dirty word

It's easy to dunk on AbstractSingletonProxyFactoryBean but let's not pretend a code base with no abstraction at all isn't also bad. It's your job to *gather* related concepts when appropriate, so your system is easier to understand and change; rather than being a collection of disparate functions and types with a lack of clarity.

Make it work, make it right, make it fast

People run in to problems with generics when they're abstracting too quickly without enough information to make good design decisions.

The TDD cycle of red, green, refactor means that you have more guidance as to what code you *actually need* to deliver your behaviour, **rather than imagining abstractions up front**; but you still need to be careful

There's no hard and fast rules here but resist making things generic until you can see that you have a useful generalisation. When we created the various `Stack` implementations we importantly started with *concrete* behaviour like `StackOfStrings` and `StackOfInts` backed by tests. From our *real* code we could start to see real patterns, and backed by our tests, we could explore refactoring toward a more general-purpose solution.

People often advise you to only generalise when you see the same code three times, which seems like a good starting rule of thumb.

A common path I've taken in other programming languages has been:

- One TDD cycle to drive some behaviour
- Another TDD cycle to exercise some other related scenarios

> Hmm, these things look similar - but a little duplication is better than coupling to a bad abstraction

- Sleep on it
- Another TDD cycle

> OK, I'd like to try to see if I can generalise this thing. Thank goodness I am so smart and good-looking because I use TDD, so I can refactor whenever I wish, and the process has helped me understand what behaviour I actually need before designing too much.

- This abstraction feels nice! The tests are still passing, and the code is simpler
- I can now delete a number of tests, I've captured the *essence* of the behaviour and removed unnecessary detail