Iteration

You can find all the code for this chapter here

To do stuff repeatedly in Go, you'll need `for`. In Go there are no `while`, `do`, `until` keywords, you can only use `for`. Which is a good thing!

Let's write a test for a function that repeats a character 5 times.

There's nothing new so far, so try and write it yourself for practice.

Write the test first

```
1  package iteration
2
3  import "testing"
4
5  func TestRepeat(t *testing.T) {
6          repeated := Repeat("a")
7          expected := "aaaaa"
8
9          if repeated != expected {
10                 t.Errorf("expected %q but got %q", expected, repeated)
11         }
12 }
```

Try and run the test

```
./repeat_test.go:6:14: undefined: Repeat
```

Write the minimal amount of code for the test to run and

Keep the discipline! You don't need to know anything new right now to make the test fail properly.

All you need to do right now is enough to make it compile so you can check your test is written well.

```
1 package iteration
2
3 func Repeat(character string) string {
4         return ""
5 }
```

Isn't it nice to know you already know enough Go to write tests for some basic problems? This means you can now play with the production code as much as you like and know it's behaving as you'd hope.

`repeat_test.go:10: expected 'aaaaa' but got ''`

Write enough code to make it pass

The `for` syntax is very unremarkable and follows most C-like languages.

```
1 func Repeat(character string) string {
2         var repeated string
3         for i := 0; i < 5; i++ {
4                 repeated = repeated + character
5         }
6         return repeated
7 }
```

Unlike other languages like C, Java, or JavaScript there are no parentheses surrounding the three components of the for statement and the braces `{ }` are always required. You might wonder what is happening in the row

```
1         var repeated string
```

as we've been using `:=` so far to declare and initializing variables. However, `:=` is simply short hand for both steps. Here we are declaring a `string` variable only. Hence, the explicit version. We can also use `var` to declare functions, as we'll see later on.

Run the test and it should pass.

Additional variants of the for loop are described here.

Refactor

Now it's time to refactor and introduce another construct `+=` assignment operator.

```
1 const repeatCount = 5
2
3 func Repeat(character string) string {
4         var repeated string
5         for i := 0; i < repeatCount; i++ {
6                 repeated += character
7         }
8         return repeated
9 }
```

`+=` called "*the Add AND assignment operator*", adds the right operand to the left operand and assigns the result to left operand. It works with other types like integers.

Benchmarking

Writing benchmarks in Go is another first-class feature of the language and it is very similar to writing tests.

```
1 func BenchmarkRepeat(b *testing.B) {
2         for i := 0; i < b.N; i++ {
3                 Repeat("a")
4         }
5 }
```

You'll see the code is very similar to a test.

The `testing.B` gives you access to the cryptically named `b.N`.

When the benchmark code is executed, it runs `b.N` times and measures how long it takes.

The amount of times the code is run shouldn't matter to you, the framework will determine what is a "good" value for that to let you have some decent results.

To run the benchmarks do `go test -bench=.` (or if you're in Windows Powershell `go test -bench="."`)

```
goos: darwin
goarch: amd64
pkg: github.com/quii/learn-go-with-tests/for/v4
10000000           136 ns/op
PASS
```

What `136 ns/op` means is our function takes on average 136 nanoseconds to run (on my computer). Which is pretty ok! To test this it ran it 10000000 times.

NOTE by default Benchmarks are run sequentially.

Practice exercises

- Change the test so a caller can specify how many times the character is repeated and then fix the code
- Write `ExampleRepeat` to document your function
- Have a look through the strings package. Find functions you think could be useful and experiment with them by writing tests like we have here. Investing time learning the standard library will really pay off over time.

Wrapping up

- More TDD practice
- Learned `for`

- Learned how to write benchmarks