Integers

You can find all the code for this chapter here

Integers work as you would expect. Let's write an `Add` function to try things out. Create a test file called `adder_test.go` and write this code.

Note: Go source files can only have one `package` per directory, make sure that your files are organised separately. Here is a good explanation on this.

Write the test first

```go
package integers

import "testing"

func TestAdder(t *testing.T) {
        sum := Add(2, 2)
        expected := 4

        if sum != expected {
                t.Errorf("expected '%d' but got '%d'", expected, sum)
        }
}
```

You will notice that we're using `%d` as our format strings rather than `%q`. That's because we want it to print an integer rather than a string.

Also note that we are no longer using the main package, instead we've defined a package named `integers`, as the name suggests this will group functions for working with integers such as `Add`.

Try and run the test

Run the test `go test`

Inspect the compilation error

`./adder_test.go:6:9: undefined: Add`

Write the minimal amount of code for the test to run and check the failing test output

Write enough code to satisfy the compiler *and that's all* - remember we want to check that our tests fail for the correct reason.

```
1 package integers
2
3 func Add(x, y int) int {
4         return 0
5 }
```

When you have more than one argument of the same type (in our case two integers) rather than having `(x int, y int)` you can shorten it to `(x, y int)`.

Now run the tests and we should be happy that the test is correctly reporting what is wrong.

`adder_test.go:10: expected '4' but got '0'`

If you have noticed we learnt about *named return value* in the last section but aren't using the same here. It should generally be used when the meaning of the result isn't clear from context, in our case it's pretty much clear that `Add` function will add the parameters. You can refer this wiki for more details.

Write enough code to make it pass

In the strictest sense of TDD we should now write the *minimal amount of code to make the test pass*. A pedantic programmer may do this

```
```

```
func Add(x, y int) int {
	return 4
}
```

Ah hah! Foiled again, TDD is a sham right?

We could write another test, with some different numbers to force that test to fail but that feels like a game of cat and mouse.

Once we're more familiar with Go's syntax I will introduce a technique called *"Property Based Testing"*, which would stop annoying developers and help you find bugs.

For now, let's fix it properly

```
func Add(x, y int) int {
	return x + y
}
```

If you re-run the tests they should pass.

Refactor

There's not a lot in the *actual* code we can really improve on here.

We explored earlier how by naming the return argument it appears in the documentation but also in most developer's text editors.

This is great because it aids the usability of code you are writing. It is preferable that a user can understand the usage of your code by just looking at the type signature and documentation.

You can add documentation to functions with comments, and these will appear in Go Doc just like when you look at the standard library's documentation.

```
// Add takes two integers and returns the sum of them.
func Add(x, y int) int {
	return x + y
}
```

Examples

If you really want to go the extra mile you can make examples. You will find a lot of examples in the documentation of the standard library.

Often code examples that can be found outside the codebase, such as a readme file often become out of date and incorrect compared to the actual code because they don't get checked.

Go examples are executed just like tests so you can be confident examples reflect what the code actually does.

Examples are compiled (and optionally executed) as part of a package's test suite.

As with typical tests, examples are functions that reside in a package's `_test.go` files. Add the following `ExampleAdd` function to the `adder_test.go` file.

```
1 func ExampleAdd() {
2         sum := Add(1, 5)
3         fmt.Println(sum)
4         // Output: 6
5 }
```

(If your editor doesn't automatically import packages for you, the compilation step will fail because you will be missing `import "fmt"` in `adder_test.go`. It is strongly recommended you research how to have these kind of errors fixed for you automatically in whatever editor you are using.)

If your code changes so that the example is no longer valid, your build will fail.

Running the package's test suite, we can see the example function is executed with no further arrangement from us:

```
1 $ go test -v
2 === RUN   TestAdder
3 --- PASS: TestAdder (0.00s)
4 === RUN   ExampleAdd
5 --- PASS: ExampleAdd (0.00s)
```

Please note that the example function will not be executed if you remove the comment `// Output: 6`. Although the function will be compiled, it won't be executed.

By adding this code the example will appear in the documentation inside `godoc`, making your code even more accessible.

To try this out, run `godoc -http=:6060` and navigate to `http://localhost:6060/pkg/`

Inside here you'll see a list of all the packages in your `$GOPATH`, so assuming you wrote this code in somewhere like `$GOPATH/src/github.com/{your_id}` you'll be able to find your example documentation.

If you publish your code with examples to a public URL, you can share the documentation of your code at pkg.go.dev. For example, here is the finalised API for this chapter. This web interface allows you to search for documentation of standard library packages and third-party packages.

Wrapping up

What we have covered:

- More practice of the TDD workflow
- Integers, addition
- Writing better documentation so users of our code can understand its usage quickly
- Examples of how to use our code, which are checked as part of our tests