Hello, World

You can find all the code for this chapter here

It is traditional for your first program in a new language to be Hello, World.

> Create a folder wherever you like
>
> Put a new file in it called `hello.go` and put the following code inside it

```
package main

import "fmt"

func main() {
	fmt.Println("Hello, world")
}
```

To run it type `go run hello.go`.

How it works

When you write a program in Go, you will have a `main` package defined with a `main` func inside it. Packages are ways of grouping up related Go code together.

The `func` keyword is how you define a function with a name and a body.

With `import "fmt"` we are importing a package which contains the `Println` function that we use to print.

How to test

How do you test this? It is good to separate your "domain" code from the outside world (side-effects). The `fmt.Println` is a side effect (printing to stdout) and the string we send in is our domain.

So let's separate these concerns so it's easier to test

```go
package main

import "fmt"

func Hello() string {
	return "Hello, world"
}

func main() {
	fmt.Println(Hello())
}
```

We have created a new function again with `func` but this time we've added another keyword `string` in the definition. This means this function returns a `string`.

Now create a new file called `hello_test.go` where we are going to write a test for our `Hello` function

```go
package main

import "testing"

func TestHello(t *testing.T) {
	got := Hello()
	want := "Hello, world"

	if got != want {
		t.Errorf("got %q want %q", got, want)
	}
}
```

Go modules?

The next step is to run the tests. Enter `go test` in your terminal. If the tests pass, then you are probably using an earlier version of Go. However, if you are using Go 1.16 or later, then the tests will likely not run at all. Instead, you will see an error message like this in the terminal:

```
$ go test
go: cannot find main module; see 'go help modules'
```

What's the problem? In a word, modules. Luckily, the problem is easy to fix. Enter `go mod init hello` in your terminal. That will create a new file with the following contents:

```
module hello

go 1.16
```

This file tells the `go` tools essential information about your code. If you planned to distribute your application, you would include where the code was available for download as well as information about dependencies. For now, your module file is minimal, and you can leave it that way. To read more about modules, you can check out the reference in the Golang documentation. We can get back to testing and learning Go now since the tests should run, even on Go 1.16.

In future chapters you will need to run `go mod init SOMENAME` in each new folder before running commands like `go test` or `go build`.

Back to Testing

Run `go test` in your terminal. It should've passed! Just to check, try deliberately breaking the test by changing the `want` string.

Notice how you have not had to pick between multiple testing frameworks and then figure out how to install. Everything you need is built in to the language and the syntax is the same as the rest of the code you will write.

Writing tests

Writing a test is just like writing a function, with a few rules

> It needs to be in a file with a name like `xxx_test.go`

> The test function must start with the word `Test`

> The test function takes one argument only `t *testing.T`

> In order to use the `*testing.T` type, you need to `import "testing"`, like we did with `fmt` in the other file

For now, it's enough to know that your `t` of type `*testing.T` is your "hook" into the testing framework so you can do things like `t.Fail()` when you want to fail.

We've covered some new topics:

`if`

If statements in Go are very much like other programming languages.

Declaring variables

We're declaring some variables with the syntax `varName := value`, which lets us re-use some values in our test for readability.

`t.Errorf`

We are calling the `Errorf` *method* on our `t` which will print out a message and fail the test. The `f` stands for format which allows us to build a string with values inserted into the placeholder values `%q`. When you made the test fail it should be clear how it works.

You can read more about the placeholder strings in the fmt go doc. For tests `%q` is very useful as it wraps your values in double quotes.

We will later explore the difference between methods and functions.

Go doc

Another quality of life feature of Go is the documentation. You can launch the docs locally by running `godoc -http :8000`. If you go to localhost:8000/pkg you will see all the packages installed on your system.

The vast majority of the standard library has excellent documentation with examples. Navigating to http://localhost:8000/pkg/testing/ would be worthwhile to see what's available to you.

If you don't have `godoc` command, then maybe you are using the newer version of Go (1.14 or later) which is no longer including `godoc`. You can manually install it with `go install golang.org/x/tools/cmd/godoc@latest`.

Hello, YOU

Now that we have a test we can iterate on our software safely.

In the last example we wrote the test *after* the code had been written just so you could get an example of how to write a test and declare a function. From this point on we will be *writing tests first*.

Our next requirement is to let us specify the recipient of the greeting.

Let's start by capturing these requirements in a test. This is basic test driven development and allows us to make sure our test is *actually* testing what we want. When you retrospectively write tests there is the risk that your test may continue to pass even if the code doesn't work as intended.

```go
package main

import "testing"

func TestHello(t *testing.T) {
    got := Hello("Chris")
    want := "Hello, Chris"

    if got != want {
        t.Errorf("got %q want %q", got, want)
    }
}
```

Now run `go test`, you should have a compilation error

```
./hello_test.go:6:18: too many arguments in call to Hello
    have (string)
    want ()
```

When using a statically typed language like Go it is important to *listen to the compiler*. The compiler understands how your code should snap together and work so you don't have to.

In this case the compiler is telling you what you need to do to continue. We have to change our function `Hello` to accept an argument.

Edit the `Hello` function to accept an argument of type string

```
func Hello(name string) string {
    return "Hello, world"
}
```

If you try and run your tests again your `hello.go` will fail to compile because you're not passing an argument. Send in "world" to make it compile.

```
func main() {
    fmt.Println(Hello("world"))
}
```

Now when you run your tests you should see something like

```
hello_test.go:10: got 'Hello, world' want 'Hello, Chris''
```

We finally have a compiling program but it is not meeting our requirements according to the test.

Let's make the test pass by using the name argument and concatenate it with `Hello,`

```
func Hello(name string) string {
    return "Hello, " + name
}
```

When you run the tests they should now pass. Normally as part of the TDD cycle we should now *refactor*.

A note on source control

At this point, if you are using source control (which you should!) I would `commit` the code as it is. We have working software backed by a test.

I *wouldn't* push to main though, because I plan to refactor next. It is nice to commit at this point in case you somehow get into a mess with refactoring - you can always go back to the working version.

There's not a lot to refactor here, but we can introduce another language feature, *constants*.

Constants

Constants are defined like so

```
const englishHelloPrefix = "Hello, "
```

We can now refactor our code

```
const englishHelloPrefix = "Hello, "

func Hello(name string) string {
    return englishHelloPrefix + name
}
```

After refactoring, re-run your tests to make sure you haven't broken anything.

It's worth thinking about creating constants to capture the meaning of values and sometimes to aid performance.

Hello, world... again

The next requirement is when our function is called with an empty string it defaults to printing "Hello, World", rather than "Hello, ".

Start by writing a new failing test

```go
func TestHello(t *testing.T) {
    t.Run("saying hello to people", func(t *testing.T) {
        got := Hello("Chris")
        want := "Hello, Chris"

        if got != want {
            t.Errorf("got %q want %q", got, want)
        }
    })
    t.Run("say 'Hello, World' when an empty string is supplied", func(t *testi
        got := Hello("")
        want := "Hello, World"

        if got != want {
            t.Errorf("got %q want %q", got, want)
        }
    })
}
```

Here we are introducing another tool in our testing arsenal, subtests. Sometimes it is useful to group tests around a "thing" and then have subtests describing different scenarios.

A benefit of this approach is you can set up shared code that can be used in the other tests.

While we have a failing test, let's fix the code, using an `if`.

```go
const englishHelloPrefix = "Hello, "

func Hello(name string) string {
    if name == "" {
        name = "World"
    }
    return englishHelloPrefix + name
}
```

If we run our tests we should see it satisfies the new requirement and we haven't accidentally broken the other functionality.

It is important that your tests *are clear specifications* of what the code needs to do. But there is repeated code when we check if the message is what we expect.

Refactoring is not *just* for the production code!

Now that the tests are passing, we can and should refactor our tests.

```go
func TestHello(t *testing.T) {
	t.Run("saying hello to people", func(t *testing.T) {
		got := Hello("Chris")
		want := "Hello, Chris"
		assertCorrectMessage(t, got, want)
	})

	t.Run("empty string defaults to 'world'", func(t *testing.T) {
		got := Hello("")
		want := "Hello, World"
		assertCorrectMessage(t, got, want)
	})

}

func assertCorrectMessage(t testing.TB, got, want string) {
	t.Helper()
	if got != want {
		t.Errorf("got %q want %q", got, want)
	}
}
```

What have we done here?

We've refactored our assertion into a new function. This reduces duplication and improves readability of our tests. We need to pass in `t *testing.T` so that we can tell the test code to fail when we need to.

For helper functions, it's a good idea to accept a `testing.TB` which is an interface that `*testing.T` and `*testing.B` both satisfy, so you can call helper functions from a test, or a benchmark (don't worry if words like "interface" mean nothing to you right now, it will be covered later).

`t.Helper()` is needed to tell the test suite that this method is a helper. By doing this when it fails the line number reported will be in our *function call* rather than inside our test helper. This will help other developers track down problems easier. If you still don't understand, comment it out, make a test fail and observe the test output. Comments in Go are a great way to add additional information to your code, or in this case, a quick way to tell the compiler to ignore a line. You can comment out the `t.Helper()` code by adding two forward slashes `//` at the beginning of the line. You should see that line turn grey or change to another color than the rest of your code to indicate it's now commented out.

Back to source control

Now we are happy with the code I would amend the previous commit so we only check in the lovely version of our code with its test.

Discipline

Let's go over the cycle again

> Write a test
>
> Make the compiler pass
>
> Run the test, see that it fails and check the error message is meaningful
>
> Write enough code to make the test pass
>
> Refactor

On the face of it this may seem tedious but sticking to the feedback loop is important.

Not only does it ensure that you have *relevant tests*, it helps ensure *you design good software* by refactoring with the safety of tests.

Seeing the test fail is an important check because it also lets you see what the error message looks like. As a developer it can be very hard to work with a codebase when failing tests do not give a clear idea as to what the problem is.

By ensuring your tests are *fast* and setting up your tools so that running tests is simple you can get in to a state of flow when writing your code.

By not writing tests you are committing to manually checking your code by running your software which breaks your state of flow and you won't be saving yourself any time, especially in the long run.

Keep going! More requirements

Goodness me, we have more requirements. We now need to support a second parameter, specifying the language of the greeting. If a language is passed in that we do not recognise, just default to English.

We should be confident that we can use TDD to flesh out this functionality easily!

Write a test for a user passing in Spanish. Add it to the existing suite.

```
	t.Run("in Spanish", func(t *testing.T) {
		got := Hello("Elodie", "Spanish")
		want := "Hola, Elodie"
		assertCorrectMessage(t, got, want)
	})
```

Remember not to cheat! *Test first*. When you try and run the test, the compiler *should* complain because you are calling `Hello` with two arguments rather than one.

```
./hello_test.go:27:19: too many arguments in call to Hello
	have (string, string)
	want (string)
```

Fix the compilation problems by adding another string argument to `Hello`

```
func Hello(name string, language string) string {
	if name == "" {
		name = "World"
	}
	return englishHelloPrefix + name
}
```

When you try and run the test again it will complain about not passing through enough arguments to `Hello` in your other tests and in `hello.go`

```
./hello.go:15:19: not enough arguments in call to Hello
    have (string)
    want (string, string)
```

Fix them by passing through empty strings. Now all your tests should compile *and* pass, apart from our new scenario

```
hello_test.go:29: got 'Hello, Elodie' want 'Hola, Elodie'
```

We can use `if` here to check the language is equal to "Spanish" and if so change the message

```go
func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    if language == "Spanish" {
        return "Hola, " + name
    }
    return englishHelloPrefix + name
}
```

The tests should now pass.

Now it is time to *refactor*. You should see some problems in the code, "magic" strings, some of which are repeated. Try and refactor it yourself, with every change make sure you re-run the tests to make sure your refactoring isn't breaking anything.

```go
    const spanish = "Spanish"
    const englishHelloPrefix = "Hello, "
    const spanishHelloPrefix = "Hola, "

    func Hello(name string, language string) string {
        if name == "" {
            name = "World"
        }

        if language == spanish {
            return spanishHelloPrefix + name
        }
        return englishHelloPrefix + name
    }
```

French

> Write a test asserting that if you pass in `"French"` you get `"Bonjour, "`

> See it fail, check the error message is easy to read

> Do the smallest reasonable change in the code

You may have written something that looks roughly like this

```go
func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    if language == spanish {
        return spanishHelloPrefix + name
    }
    if language == french {
        return frenchHelloPrefix + name
    }
    return englishHelloPrefix + name
}
```

`switch`

When you have lots of `if` statements checking a particular value it is common to use a `switch` statement instead. We can use `switch` to refactor the code to make it easier to read and more extensible if we wish to add more language support later

```
func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    prefix := englishHelloPrefix

    switch language {
    case "French":
        prefix = frenchHelloPrefix
    case "Spanish":
        prefix = spanishHelloPrefix
    }

    return prefix + name
}
```

Write a test to now include a greeting in the language of your choice and you should see how simple it is to extend our *amazing* function.

one...last...refactor?

You could argue that maybe our function is getting a little big. The simplest refactor for this would be to extract out some functionality into another function.

```go
const (
    french  = "French"
    spanish = "Spanish"

    englishHelloPrefix = "Hello, "
    spanishHelloPrefix = "Hola, "
    frenchHelloPrefix  = "Bonjour, "
)

func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }

    return greetingPrefix(language) + name
}

func greetingPrefix(language string) (prefix string) {
    switch language {
    case french:
        prefix = frenchHelloPrefix
    case spanish:
        prefix = spanishHelloPrefix
    default:
        prefix = englishHelloPrefix
    }
    return
}
```

A few new concepts:

- In our function signature we have made a *named return value* `(prefix string)`.
- This will create a variable called `prefix` in your function.
  - It will be assigned the "zero" value. This depends on the type, for example `int`s are 0 and for `string`s it is `""`.
    - You can return whatever it's set to by just calling `return` rather than `return prefix`.
  - This will display in the Go Doc for your function so it can make the intent of your code clearer.
- `default` in the switch case will be branched to if none of the other `case` statements match.
- The function name starts with a lowercase letter. In Go, public functions start with a capital letter and private ones start with a lowercase. We don't want the internals of our algorithm to be exposed to the world, so we made this function private.
- Also, we can group constants in a block instead of declaring them each on their own line. It's a good idea to use a line between sets of related constants for readability.

Wrapping up

Who knew you could get so much out of `Hello, world`?

By now you should have some understanding of:

Some of Go's syntax around

- Writing tests
- Declaring functions, with arguments and return types
- `if`, `const` and `switch`
- Declaring variables and constants

The TDD process and *why* the steps are important

> *Write a failing test and see it fail* so we know we have written a *relevant* test for our requirements and seen that it produces an *easy to understand description of the failure*

> Writing the smallest amount of code to make it pass so we know we have working software

> *Then* refactor, backed with the safety of our tests to ensure we have well-crafted code that is easy to work with

In our case we've gone from `Hello()` to `Hello("name")`, to `Hello("name", "French")` in small, easy to understand steps.

This is of course trivial compared to "real world" software but the principles still stand. TDD is a skill that needs practice to develop, but by breaking problems down into smaller components that you can test, you will have a much easier time writing software.

Previous
Install Go

Next
Integers

Last updated 5 months ago