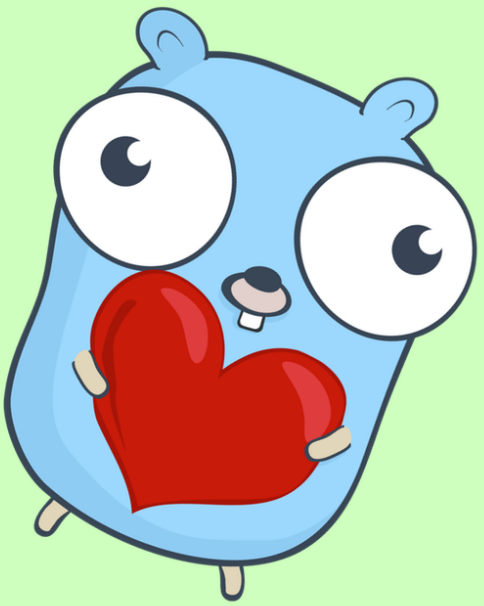
For the Love of Go

John Arundel

September 4, 2022

FOR THE LOVE OF GO



“Personable, human, and funny”

—Kevin Cunningham

JOHN ARUNDEL

Contents

Praise for *For the Love of Go*

A great way to dive into Go!
—Max VelDink

One of the best technical books I have read in a very long time.
—Paul Watts

John is both a superb engineer and, just as importantly, an excellent teacher / communicator.
—Luke Vidler

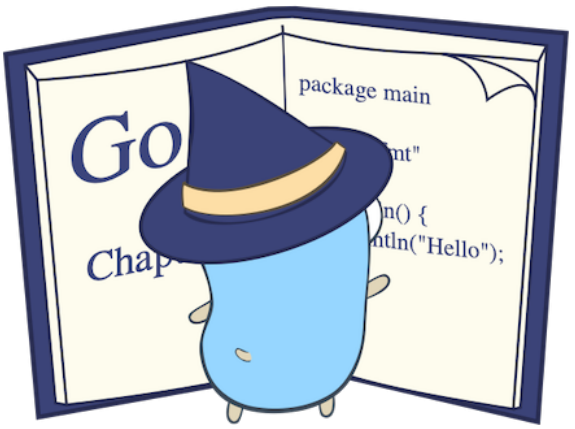
A fantastic example of good technical writing. Clear, concise and easily digestible.
—Michael Duffy

This book's writing style feels as if John is speaking to you in person and helping you along the way.
—@rockey5520

Very well written, friendly, and informative.
—Chris Doyle

John's writing is personable, human and funny—his examples are realistic and relevant. The test-driven instruction teaches Go in a deep, meaningful and engaging way.
—Kevin Cunningham

Introduction



Hello, and welcome to learning Go! It's great to have you here.

What's this?

This book is an introduction to the Go programming language, suitable for complete beginners. If you don't know anything about Go yet, or programming, but would like to learn, you're in the right place.

If you do already have some experience with Go, or with programming in other languages, don't worry: this book is for you too! You'll learn some ideas, techniques, and ways of tackling problems that even many advanced Go programmers don't know. I hope to also help you fill in a few gaps in your knowledge that you may not even be aware of.

What version of Go does it cover?

This book uses Go 1.19, and all the code samples have been tested against at least that version. However, Go puts a strong emphasis on backward compatibility, so all the code in this book will still work perfectly well with later Go 1.x versions.

What you'll need

You'll need to install Go on your computer, if you don't have it already. Follow the instructions on the Go website to download and install Go:

- https://go.dev/learn/

You'll also need at least Go version 1.16 to run the code examples in this book. Run the following command to see what version of Go you have:

```
go version
```

```
go version go1.19 darwin/amd64
```

While all you need to write and run Go programs is a terminal and a text editor, you'll find it very helpful to use an editor that has specific support for Go. For example, Visual Studio Code has excellent Go integration.

There's a dedicated Go editor called GoLand that you may come across: I *don't* recommend this, at least for beginners. Although GoLand is a very powerful tool for professional developers, it makes some non-obvious choices about how to actually represent your Go code on screen, and this can be confusing for newcomers. By all means try out GoLand to see if you like it, but I'd recommend starting with Visual Studio Code and getting used to that first.

While not essential, it's a great idea to use some kind of *version control* software (for example, Git) to track and share your source code. I won't go into the details of how to install and use Git in this book, but if you're not already familiar with it, I recommend you learn at least the basics. Go to GitHub to find out more.

Where to find the code examples

There are dozens of challenges for you to solve throughout the book, each designed to help you test your understanding of the concepts you've just learned. If you run into trouble, or just want to check your code, each challenge is accompanied by a complete sample solution, with tests.

All these solutions are also available in a public GitHub repo here:

- https://github.com/bitfield/ftl-code

Each listing in the book is accompanied by a number (for example, Listing 1.1, and you'll find the solution to that exercise in the numbered folder of the repo.

What you'll learn

By reading through this book and completing the exercises, you'll learn:

- How to write *tests* in Go and how to develop projects guided by tests

- How to manage data in Go using built-in types, user-defined struct types, and collections such as maps and slices
- How to write and test *functions*, including functions that return multiple results and error values
- How to use *objects* to model problems in Go, and how to add behaviour to objects using *methods*
- How to use *pointers* to write methods that modify objects, and how to use *types* and *validation* to make your Go packages easy to use

The love of Go

Go is an unusually fun and enjoyable language to write programs in, and I hope to communicate something of my own love of Go to you in this book. It's a relatively *simple* language, which isn't the same as easy, of course. Programming is not easy, and no language makes it easy. Some just make it a little more fun than others.

Go lacks many things that other programming languages have, and that's no accident. In some ways, the fact that Go doesn't have many features is its best feature. It has *enough* features to do everything we need to do in building software, but no more. The fewer features there are, the less we have to learn, and the more quickly we can get on to *using* our new tool to do interesting things.

Because each little bit of code doesn't really do much by itself, it's quite easy to read Go programs, even if you haven't got much experience with Go. As programmers, we don't need to worry about choosing among many possible ways to implement the same logic. This leaves us free to think about the much more important issue of how to *structure* our programs in clear, simple ways that are easy for readers to understand.

Go is humble, and that's a good attitude for us to adopt, too. It doesn't claim to be the world's greatest programming language, the most advanced, or the most pure, elegant, and theoretically appealing. It isn't. The designers of Go wanted to maximise *productivity*, not elegance, and that explains a lot.

Similarly, the programs we write with Go should be simple, straightforward, and pragmatic. If they're elegant too, that's nice, but it's not the goal. And because Go programs tend to be simple and easy to understand, it's easy to change them later, so we don't need to over-engineer everything at the start. We just build what's necessary to get the job done and leave it at that.

We try to solve the problem we *have*, not the problem that might face us next week, or next year. Rather than blindly applying coding dogmas like "always write short functions", regardless of the situation, Go programmers are more interested in what makes the *program* clearer.

Programming with confidence

It's very important that the programs we write be *correct*. That is, they should do what they're intended to do. Indeed, if the program is not correct, hardly anything else about it matters.

One way to improve our confidence in the program's correctness is to write *tests* for it.

A test is also a program, but it's a program specifically designed to run *another* program with various different inputs, and check its result. The test verifies that the program actually behaves the way it's supposed to.

For example, if you were considering writing a function called `Multiply`, and you wanted to be able to check that it worked, you might write a program that calls `Multiply(2, 2)` and checks that the answer is `4`. That's a test!

And it turns out that it's a good idea to write the test even before you start writing the function it tests. It might seem a little odd to do things this way round, if you're not used to it. But it actually makes a lot of sense.

By writing the test first, you are forced to think about what the program actually needs to do. You also have to design its interface (the way users interact with the program). If it's really awkward to call your function, or it has a name that doesn't make sense in the context where it's used, all of these things will be obvious when you write your test. In effect, you're being your own first user.

And it also means you know when to stop coding, because when the test starts passing, you're done!

So writing tests is not just about gaining confidence that your code is correct, although that's definitely useful. It's also a powerful process for designing well-structured programs with good APIs. A phrase I like that describes this process is "development, *guided by tests*".

What makes this book different?

Most Go books will start by telling you about the basic syntax of the language, variables, data types, and so on. That's important stuff to know, and we'll cover it in this book too. But because I want you to get thoroughly comfortable with the process of software development guided by tests, we're going to be doing that right from the start.

In order to do this, we'll *use* some Go concepts that won't be familiar to you yet: functions, variables, the `testing` package, and so on. Don't worry about understanding everything that's going on at first: we'll get to the details later. For the first few chapters, just relax and enjoy the ride, even if you come across some things that seem a little mysterious.

In fact, this is often the situation we find ourselves in as programmers: we *have* some existing code that we don't necessarily totally understand, but we will try to figure out

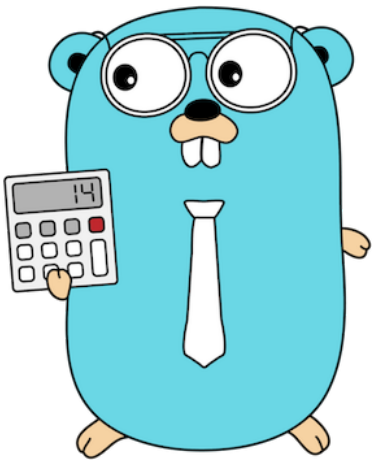
just enough to solve the problem at hand. So this will be good practice for you, and everything will be explained in its right place.

How to use this book

Throughout this book we'll be working together to develop a project in Go. Each chapter introduces a new feature or concept, and sets you some goals to achieve. Goals are marked with **GOAL** in bold. (Don't worry if you're not sure how to start. There'll usually be some helpful suggestions following the goal description.)

So let's get started!

1. Testing times



It's your first day as a Go developer at Texio Instronics, a major manufacturer of widgets and what-nots, and you've been assigned to the Future Projects division to work on a very exciting new product: a Go-powered electronic calculator. Welcome aboard.

In this exercise, all we need to do is make sure our Go environment is set up and everything is working right. We're going to create a new Go project for the calculator package, add some test code to it, and run the test.

Creating a new project

Every Go project needs two things: a *folder* on disk to keep its source code in, and a `go.mod` file which identifies the *module*, or project name.

If you don't have one already, I recommend you create a new folder on your computer to keep your Go projects in (this can be in any location you like). Then, within this folder, create a subfolder named `calculator`.

Next, start a shell session using your terminal program (for example, the macOS Terminal app) or your code editor. Set your working directory to the project folder using the `cd` command (for example, `cd ~/go/calculator`).

Now run the following command in the shell:

```
go mod init calculator
```

```
go: creating new go.mod: module calculator
```

You should find that a new file has been created named `go.mod`, with the following contents (or something very similar):

```
module calculator

go 1.19
```

(Listing 1.1)

Go organises code into units called *modules*, and each module needs to be in a separate folder (you can't have modules within modules). Each module needs a `go.mod` file which tells Go that this folder contains a module, and what the name of the module is (in this case, it's `calculator`).

So we should now have a folder structure something like this:

```
calculator/
    go.mod
```

All clear? Great! We'll be creating a number of new Go projects throughout this book, using the same steps shown in this section: create a new folder, and inside the folder, run the command `go mod init MODULE_NAME`, where `MODULE_NAME` represents the name we want to give the project.

Next, we'll see how to create some Go files and run tests.

Creating Go files

So that we don't have to start entirely from scratch, a helpful colleague at Texio Instronics has sent us some Go code that implements part of the calculator's functionality, plus a test for it. In this section we'll add that code to the project folder.

First, using your code editor, create a file in the `calculator` folder named `calculator.go`. Copy and paste the following code into it:

```
// Package calculator does simple calculations.
package calculator

// Add takes two numbers and returns the result of adding
// them together.
func Add(a, b float64) float64 {
    return a + b
}
```

(Listing 1.1)

If it's not easy for you to copy and paste the code from this book, clone the GitHub repository instead and copy from there:

- https://github.com/bitfield/ftl-code

Don't worry about understanding all the code here for the moment; we'll cover this in detail later. For now, just paste this code into the `calculator.go` file and save it.

Next, create another file named `calculator_test.go` containing the following:

```
package calculator_test

import (
    "calculator"
    "testing"
)

func TestAdd(t *testing.T) {
    t.Parallel()
    var want float64 = 4
    got := calculator.Add(2, 2)
    if want != got {
        t.Errorf("want %f, got %f", want, got)
    }
}
```

(Listing 1.1)

Here's the folder structure we should end up with:

```
calculator/
    calculator.go
    calculator_test.go
    go.mod
```

Again, you don't need to fully understand what this code is doing yet (but you might be able to guess some of it). Soon, you'll not only understand how this works, but you'll be writing your own tests.

But before we get to that, we need to make sure that our Go environment is properly installed and working, the project is set up correctly, and that we can successfully run tests for a project.

Running the tests

Even though it's pretty small, this is a fully-functioning Go project, and as all good projects should, it contains some tests. The next thing for us to do is to *run* them.

GOAL: Run the tests for this project and make sure they pass. The following instructions will show you how to do that.

Still in the `calculator` folder, run the command:

```
go test
```

If everything works as it should, you will see this output:

```
PASS
ok      calculator      0.234s
```

This tells us that all the tests passed in the package called `calculator`, and that running them took 0.234 seconds. (It might take a longer or shorter time on different computers, but that's okay.)

If you see test failure messages, or any other error messages, there may be a problem with your Go setup. Don't worry, everybody runs into this kind of issue at first. Try Googling the error message to see if you can figure out what the problem is, and check the installation instructions on the Go website here:

- https://go.dev/learn/

Once the test is passing, you can move on!

Try using your editor's Go support features to run the test directly in the editor. For example, in Visual Studio Code, there's a little *Run test* link above each test function, and a *Run package tests* link at the top of the file. Find out how to do this in the editor you're using, and run the test successfully.

Formatting code with `gofmt`

The next thing to do is ensure that our Go code is formatted correctly. All Go code is formatted in a standard way, using a tool that comes with Go called `gofmt` (pronounced, in case you were wondering, "go-fumpt"). In this exercise we'll see how to use it to check and reformat Go code (if necessary).

First, let's introduce a deliberate formatting error into the code, so that we can try to find it with `gofmt`. Edit the `calculator.go` file and look for this line:

```
func Add(a, b float64) float64  {
```

(Listing 1.1)

Insert an extra space before the word `Add`, so that the line looks like this:

```
func  Add(a, b float64) float64  {
```

GOAL: Run `gofmt` to check the formatting and find out what is wrong:

```
gofmt -d calculator.go
```

```
...
 // Add takes two numbers and returns the result of adding
 // them together.
-func  Add(a, b float64) float64 {
+func Add(a, b float64) float64 {
        return a + b
 }
```

Ignore the preamble for the moment, and focus on the lines beginning with `-` and `+`:

```
-func  Add(a, b float64) float64 {
+func Add(a, b float64) float64 {
```

`gofmt` is telling us what it wants to do. It wants to remove (`-`) the line that has the bogus extra space before the `Add` function, and replace it (`+`) with a line where the space has been removed.

NOTE: If you're using Windows, you may see an error like this:

```
computing diff: exec: "diff": executable file not found in
%PATH%
```

This is because the `gofmt` tool relies on there being a `diff` program in your `PATH` that it can use to generate a visual diff of the file. If you don't have `diff` installed, or if it's not in your default `PATH`, this won't work. Don't worry, though: this isn't a serious problem, and it won't stop you being able to develop with Go; it just means you won't be able to use the `gofmt -d` command to check your formatting.

You may find this Stack Overflow discussion helpful for resolving this issue:

- `gofmt -d` error on Windows

Fixing format with `gofmt -w`

We could make the necessary formatting change using an editor, but there's no need: `gofmt` can do it for us automatically.

GOAL: Run:

```
gofmt -w calculator.go
```

This will reformat the file in place. Check that it's worked by running `gofmt -d` on the file again. Now there should be no output, because it's already formatted correctly.

If you're using an editor that has Go support, set up the editor to automatically run your code through `gofmt` every time you save it. This will help you avoid forgetting to format your code.

Just for fun, try experimenting with `gofmt` a little. Try formatting code in different ways and see what changes `gofmt` makes to it. It's interesting to note what it does and doesn't change.

If you like `gofmt`, there's a similar tool named `gofumpt` that works in much the same way, but does slightly more reformatting than the standard `gofmt` tool.

Functions in Go

Let's take a closer look now at the `Add` function and identify some important parts of function syntax in Go:

```
func Add(a, b float64) float64 {
    return a + b
}
```

(Listing 1.1)

This is a *function declaration*. It tells Go the name of the function we're going to write, any *parameters* that it takes as its input, and any *results* that it returns. Then come some statements within curly braces, known as the *body* of the function. That's the bit that actually does something.

A function declaration is introduced by the keyword `func` (meaning "here comes a function!"). Following `func` is the name of the function (`Add`), and a list of its parameters in parentheses `(a, b float64)`.

Then comes the list of *results* it returns (if any). Here that's a single `float64` value, and it doesn't have a name, so we write just `float64` before the opening curly brace (`{`).

Inside the curly braces is a list of statements that form the *body* of the function. In our example, there's only one statement: `return a + b`. This ends the function, returning the value of `a + b` as its `float64` result.

We'll look at functions in much more detail later on in the book, in the chapter on "Fun with functions". For now, though, let's continue working on the calculator project.

A failing test

Now that our development environment is all set up, a colleague needs our help. She has been working on getting the calculator to subtract numbers, but there's a problem: the test is not passing. Let's see what we can do to fix it.

Copy and paste the following code into the `calculator_test.go` file, after the `TestAdd` function:

```
func TestSubtract(t *testing.T) {
    t.Parallel()
```

```
    var want float64 = 2
    got := calculator.Subtract(4, 2)
    if want != got {
        t.Errorf("want %f, got %f", want, got)
    }
}
```

(Listing 3.1)

Here's the code she's using to implement the `Subtract` function. Copy this into the `calculator.go` file, after the `Add` function:

```
// Subtract takes two numbers a and b, and
// returns the result of subtracting b from a.
func Subtract(a, b float64) float64 {
    return b - a
}
```

(Listing 3.1)

Now, apparently this code is not working, so our challenge is to figure out why not, and fix it. Fortunately, we have a failing test to help us.

Save the modified files, and run the tests with the following command:

```
go test
```

```
--- FAIL: TestSubtract (0.00s)
    calculator_test.go:22: want 2.000000, got -2.000000
FAIL
exit status 1
FAIL    calculator      0.178s
```

There's a lot of helpful information here, so let's learn how to interpret this test output. First of all, something is failing, because we see the word `FAIL`. The next thing we want to know is what's failing, and why.

We have two tests, and only one of them is failing at the moment. Which one? The output from `go test` tells us the name of the failing test:

```
--- FAIL: TestSubtract (0.00s)
```

And that's a good start, but we can learn still more. The output tells us the exact source file and line number where the test failure happened:

```
calculator_test.go:22
```

Let's have another quick look at the test to find line 22:

```
func TestSubtract(t *testing.T) {
    t.Parallel()
```

```
    var want float64 = 2
    got := calculator.Subtract(4, 2)
    if want != got {
        t.Errorf("want %f, got %f", want, got)
    }
}
```

(Listing 3.1)

It's the line that calls `t.Errorf` to trigger the test failure, which makes sense. And why was that statement executed? Only because the condition in the `if` statement was true: in other words, `got` was not equal to `want`.

And that's reinforced by the failure message:

```
want 2.000000, got -2.000000
```

Now we have all the information we need to fix the problem. Before we tackle that, though, let's go through the test in detail, breaking down exactly what it does.

The `testing` package

You might have noticed that in the `calculator_test.go` file, there's an `import` statement near the top of the file that looks like this:

```
import (
    "calculator"
    "testing"
)
```

In Go, we have to explicitly *import* any packages that we want to use, including, in this case, the `calculator` package itself. Since we're testing it, we need to call its functions, and we can't do that unless we have imported it.

Similarly, we are using functions from the `testing` package, so we import that too. What is `testing`? It's a package in the Go standard library that relates to testing, as you might have guessed.

Go has built-in support for writing tests. Unlike with some languages, we don't need any extra framework or third-party package in order to test Go code. There are plenty of such packages if you'd like to experiment with them, but I recommend the standard, built-in `testing` package to start with.

We'll be using the `testing` package a lot, so it's worth taking a little time to read the documentation and get familiar with it:

- https://pkg.go.dev/testing

The signature of test functions

We can also see that `TestSubtract` is a Go function (because it's introduced by the `func` keyword):

```
func TestSubtract(t *testing.T) {
```

Each test in a Go project is a function, just like the `Add` and `Subtract` functions we've already seen. But in order for Go to recognise a particular function as a *test*, there are a few special requirements.

First, it must be in a file whose name ends in `_test.go` (there can be multiple tests in the same file). Go uses this filename pattern to recognise source code files that contain tests.

Also, the name of a test function must begin with the word `Test`, or Go won't recognise it as a test. It also has to accept a parameter whose type is `*testing.T`. Don't worry about exactly *what* this is for the moment; think of it as the thing that allows us to control the outcome of the test.

Finally, test functions don't return anything (there's a blank between the closing parenthesis and the opening curly brace, where the result list would normally go). So this is the signature of a test function in Go.

The test body

So what exactly is `TestSubtract` doing? Let's take a closer look at the body of the function. Here it is again, and we'll look at it line by line:

```
func TestSubtract(t *testing.T) {
    t.Parallel()
    var want float64 = 2
    got := calculator.Subtract(4, 2)
    if want != got {
        t.Errorf("want %f, got %f", want, got)
    }
}
```

(Listing 3.1)

Firstly, the `t.Parallel()` statement is a standard prelude to tests: it tells Go to run this test concurrently with other tests, which saves time.

The following statement sets up a variable named `want`, to express what it *wants* to receive from calling the function under test (`Subtract`):

```
var want float64 = 2
```

Then it calls the `Subtract` function with the values 4 and 2, storing the result into another variable `got`:

```
got := calculator.Subtract(4, 2)
```

We knew that the test must *call* the `Subtract` function at some point, if it's to test anything about its behaviour. And this is where that happens. We call `Subtract` with 4 and 2, asking in effect "What is the result of subtracting 2 from 4?"

Whatever is returned by `Subtract` is what it thinks the result of that subtraction should be, and we've stored that value in the `got` variable.

Now we have two variables, `want` and `got`. The `want` variable contains the value we expect from the subtraction—what we *want* the function to return when it's working correctly. And `got` contains what it *actually* returned.

You can guess what's coming next: we're going to compare the two. If they're the same, then we *got* the result we *wanted*, so everything's fine—test pass. We already know that's not the case at the moment, though, which is why the test is failing.

This is a very common way of writing tests: what we might call the *want-and-got* pattern. In other words, set up a variable `want` containing the correct result of the operation, call the function under test, and store its results in a variable `got`. Then compare the two.

That's what this `if` statement does:

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

We'll look at this statement and the call to `t.Errorf` in a bit more detail shortly, but before that, let's see what we can do to fix the function and get this test passing. Over to you!

The function under test

We know the test is failing, so `want` is *not* equal to `got`. When we subtract 2 from 4, we *want* 2 as the answer. What we *got* though, was not 2, but *negative* 2:

```
want 2.000000, got -2.000000
```

It's as though, instead of subtracting 2 from 4, the function had subtracted 4 from 2! Let's look at the code for the `Subtract` function to see if we can work out why.

Here it is (in `calculator.go`):

```
func Subtract(a, b float64) float64 {
    return b - a
}
```

(Listing 3.1)

So, can you fix the problem? Have a try and see what you can do.

GOAL: Get `TestSubtract` passing!

If you spot a problem in the `Subtract` function, try altering the code to fix it. Run `go test` again to check that you got it right. If not, try calling `Subtract` with some different values, and see if you can figure out what it's doing wrong.

When the test passes, you can move on! If you get stuck, have a look at my solution in Listing 3.2.

Finding the correct code for `Subtract` is not actually the point here, although it's nice to have it working. What we're really learning is how *tests* work, and most importantly, how tests can help us find and fix bugs in our code.

`if` statements

Let's take a closer look now at a specific line of the test code to see how it works:

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

This is a useful kind of Go statement, called an `if` statement. It can make the program take a certain *action*, depending on whether some *condition* happens to exist at the time. For example, in this case what it does is to take the action "fail the test" (by calling `t.Errorf`) if the condition "`want` is different to `got`" is true.

An `if` statement begins with the keyword `if`, followed by some expression that evaluates to either `true` or `false`.

We've seen expressions that evaluate to numbers before (for example, `3 + 1` evaluates to 4). Here, we have the expression `want != got`. The result of this expression is not a *number*, but one of two possible values: `true` or `false`.

These are called *boolean* values, after the mathematician George Boole, who showed that we can manipulate them arithmetically, in some sense, just like numbers. There's a special data type in Go for boolean values, named `bool`.

Conditional expressions

The expression after the `if` keyword is what determines whether or not the code inside the indented block will be executed. The result of evaluating this expression must be a boolean value, as we've seen; that is, it must be either `true` or `false`.

If it's `true`, then the indented statements will be executed. In this case, that's the call to `t.Errorf` that causes the test to fail. On the other hand, if the expression evaluates to

`false`, then nothing happens; the program just moves on to the next line after the `if` statement.

The expression in our example is this:

```
want != got
```

We call this the *conditional expression* for the `if` statement, because it specifies the condition under which the indented code should be executed.

Note that the indentation here helps us visualise what's going on:

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

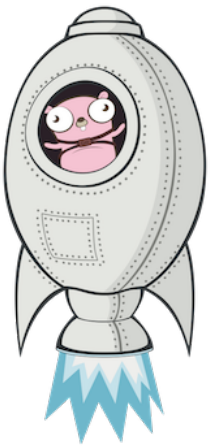
Normally, execution proceeds through a function in a straight line "downwards", if you like, following the left margin of the code. But the code inside the `if` block is indented, showing that the path of execution changes if the condition is true. Instead of a straight line down the left margin, it zigzags inward, to the right.

Takeaways

- The `go mod init` command initialises a new Go project folder.
- The `go test` command runs tests for a Go package.
- The `gofmt` command will check Go code for standard formatting and fix it if necessary.
- A *function declaration* announces the name of a new function, and lists the parameters it takes, along with any results it returns.
- Whenever a Go test fails, it shows the exact file and line where the failure happened, as well as what failed.
- The `import` keyword tells Go what other packages we want to use in this particular Go file.
- The `testing` package is part of Go's standard library, and it provides everything we need to write and run tests.
- The names of test functions begin with the word `Test`, and they take a `*testing.T` parameter which lets us control the test execution.
- Use the *want and got* pattern to structure tests: express what result you `want`, and then compare it with what you `got` from calling the function under test.
- We can call `t.Errorf` to cause the test to fail with a message, and if we don't do that, the test will pass by default.

- An `if` statement is used to take some action (for example, fail the test) if some condition is true (for example, `want` doesn't match `got`).

2. Go forth and multiply



Excellent work. We now have a calculator that can add and subtract. That's a great start. Let's build a new feature now: *multiplication*.

Designing a new feature, guided by tests

Up to now we've been running existing tests and modifying existing code. Now we're going to write both the test *and* the function it's testing.

We'll start with the test, so here's the first challenge:

GOAL: Write a test for a function `Multiply` that takes two numbers as parameters, multiplies them together, and returns a single number representing the result.

Where should we start? Well, this is a test, so start by editing the `calculator_test.go` file.

We're adding a new test, so we'll need to write a new function in this file. What should we call it?

Well, we know that test functions in Go have to have a name that starts with `Test`, or Go won't recognise them as tests. The rest of the name is up to us, though. How about `TestMultiply`?

We already have everything we need to write the test, based on what we've learned in the previous chapter. But we needn't write it from scratch; after all, we have two very

similar tests already to use as a model.

Let's start by copying the `TestAdd` function, pasting it at the end of the test file, and making some changes to it.

First, we need to change the *name* of the test, because Go doesn't allow us to have two functions with the same name in the same package. We'll change the name to `TestMultiply`.

The second thing we'll need to change is the call to `calculator.Add`. That should now be `calculator.Multiply`. Let's change the arguments to the function, too, making them 3 and 3.

Finally, we need to update the value assigned to `want`. What do we want the result of multiplying 3 and 3 to be? I think it's 9, but you might want to check my math here.

Here's the result of making these changes:

```
func TestMultiply(t *testing.T) {
    t.Parallel()
    var want float64 = 9
    got := calculator.Multiply(3, 3)
    if want != got {
        t.Errorf("want %f, got %f", want, got)
    }
}
```

(Listing 4.1)

Let's run the tests to see how we're doing so far:

```
go test
```

This gives us a compiler error:

```
undefined: calculator.Multiply
```

This is totally okay, because we expected it. We haven't *written* the `Multiply` function yet, so no wonder the compiler can't find it. Ignore this error for now; it will go away as soon as we've provided a definition of `Multiply`.

Testing a null implementation

Now we're ready to take the next step, to get to a *failing* test. That sounds a bit strange: shouldn't we be aiming for a *passing* test instead?

Well, yes, eventually. But there's an important step to take first. The job of the `TestMultiply` function is to detect bugs in `Multiply`, if there are any. How do we know it will actually do that?

For example, if we wrote the correct implementation of `Multiply` now and ran the test again, presumably it would pass. But there could easily be some mistake in the *test* that makes it pass by accident. As we've seen, tests in Go pass by default, if we don't do anything to make them fail.

What could be wrong with the test? Well, we could have the wrong `want` value. Suppose, by some unlucky chance, a buggy `Multiply` function returned the exact wrong value that we placed in the `want` variable. The test would pass, even though the function doesn't work.

Or we might write the wrong conditional expression in the `if` statement. For example, this kind of mistake is easy to make:

```
if want == got {
    ... // fail
}
```

By writing `==` instead of `!=`, we've made the test do the exact opposite of what it should. If `Multiply` returns the correct answer, the test will fail!

So how *can* we prove to ourselves that the test correctly detects when `Multiply` is not working? One way is to deliberately write a version of `Multiply` that we *know* doesn't work, and see if the test detects it.

GOAL: Write an incorrect version of `Multiply`, and show that the test fails as a result.

We'll be adding the new function to the `calculator.go` file, and it will look fairly similar to the existing `Add` and `Subtract` functions.

How exactly should we write the function *incorrectly*, to verify the test? Well, it really doesn't matter, because literally any other result than the correct one will do for our purposes.

The easiest thing to do is probably to return zero, no matter what the inputs were:

```
func Multiply(a, b float64) float64 {
    return 0
}
```

This is what we call a *null implementation* of the function. It compiles, we can call it from a test and get a result. But since the function doesn't really do any calculation, the result is almost guaranteed to be wrong.

Remember, we're not interested in `Multiply` right now. Our task is solely to show that the *test* we've written for `Multiply` can correctly detect bugs in it. And the fact that it always returns zero is a pretty major bug!

If we can't detect that, we're off to a bad start. Let's try:

```
go test
```

```
--- FAIL: TestMultiply (0.00s)
```

```
calculator_test.go:31: want 9.000000, got 0.000000
```

Great. That's what we wanted. We've proved that `TestMultiply` can detect at least *this* bug, so it's doing *something*.

If we hadn't done that, we'd really have no way of knowing whether `TestMultiply` could detect any bugs at all—in other words, whether it really *tests* anything.

Now that we know it does, we can go ahead and write the *real* implementation of `Multiply`, with some confidence that the test will tell us whether or not we've got it right.

Writing the real implementation

The last step in this process is relatively easy. All we need to do is change the `Multiply` function so that it returns the *correct* answer, instead of zero. And, as we've seen, we can check the code by running the test. As soon as it passes, we're done!

If you get stuck, see one possible solution in Listing 4.1.

Now that we've successfully designed and implemented a new calculator feature, guided by tests, it's time to think a little more about testing more complicated behaviour.

The `Add`, `Subtract`, and `Multiply` functions are so simple that it's difficult (though not at all impossible, for a determined programmer) to get them wrong. If `TestAdd` passes for one set of inputs, for example, it's hard to imagine other inputs that would make it fail.

But this doesn't apply to some other functions, where it's quite likely that they could work correctly for some inputs, but not for others. In this situation it will be a good idea to test multiple *cases* for each function: that is to say, different inputs with different expected results.

For example, the current version of `TestAdd` tests calling `Add(2, 2)` and checks that the result is `4`. We could write a new function `TestAdd1and1`, for example, that calls `Add(1, 1)` and expects `2`. But this would be tedious for many different pairs of inputs.

Introducing test cases

Instead, the most elegant solution would be to define a number of different *test cases*—each representing a pair of inputs, together with the corresponding expected output—and then loop over them, calling `Add` with the inputs and checking the output against that expected. If *any* test case fails, the whole test fails. These are sometimes called *table tests*, because they represent the test data as a table of inputs together with their expected results.

Let's see what that might look like. First, it will be convenient to define a `struct` type

to represent the test case. `struct` is Go's name for a structured data type: that is, something that contains multiple different bits of information, united into a single record.

Don't worry if this isn't familiar; we will cover this topic in more detail later. What we're really concerned with here is the test logic, and this is just a way of setting up our test cases so that we can get to that.

Let's do this inside the `TestAdd` function, since the test case will usually be specific to the function we're testing:

```
func TestAdd(t *testing.T) {
    t.Parallel()
    type testCase struct {
        a, b float64
        want float64
    }
    ...
}
```

(Listing 5.1)

A slice of test cases

If you already have some experience of Go, you may be familiar with the notion of a *slice*: it's just Go's name for a *bunch* of things. In some languages this is called an *array*, but it's the same idea whatever we call it: a sequence of values of the same type.

Again, we'll talk more about slices and other data types later on in this book. But for now, we needn't worry about the details; we're just going to *use* a slice to represent a bunch of test cases.

Rather than write lots of very similar tests that only differ in their inputs, we would like to write just *one* test that tests many cases. For each case, we'll do what we did before: call the `Add` function with the inputs `a` and `b`, and check its result against `want`.

So having defined the data type that we're going to use for our test cases (`testCase`), let's actually provide some data of this kind. To do this, we'll create a variable `testCases`, and assign some values to it.

```
func TestAdd(t *testing.T) {
    t.Parallel()
    type testCase struct {
        a, b float64
        want float64
    }
    testCases := []testCase{
```

```
        {a: 2, b: 2, want: 4},
        {a: 1, b: 1, want: 2},
        {a: 5, b: 0, want: 5},
    }
    ...
}
```

(Listing 5.1)

Here, we're creating a variable called `testCases`, whose value is a slice of `testCase` structs containing three elements.

Each element of this slice, then, is an instance of the `testCase` struct type we defined earlier. It has the fields `a`, `b`, and `want`, representing the inputs and expected outputs of the test case.

Next, we need to test the result of `Add` for each of our cases. That involves repeating a section of code multiple times, each time using a different test case.

Looping over test cases

Now that we have the test cases, we can write the code to loop over them. We'll be using the `range` operator to do this (again, don't worry if this is unfamiliar right now: we'll come back to it later on). We're still inside the `TestAdd` function, and having previously declared our `testCases` slice, we can now proceed to use it in a `for` statement, introducing a loop:

```
func TestAdd(t *testing.T) {
    type testCase struct {
        a, b float64
        want float64
    }
    testCases := []testCase{
        {a: 2, b: 2, want: 4},
        {a: 1, b: 1, want: 2},
        {a: 5, b: 0, want: 5},
    }
    for _, tc := range testCases {
        got := calculator.Add(tc.a, tc.b)
        if tc.want != got {
            t.Errorf("Add(%f, %f): want %f, got %f", tc.a, tc.b,
                tc.want, got)
        }
```

```
    }
}
```

(Listing 5.1)

Inside the loop, `tc` will be the current `testCase` struct, so `tc.a` and `tc.b` will be the inputs to `Add`, and `tc.want` will be the expected result. Run this test and make sure it still passes—it should, shouldn't it?

GOAL: Rewrite `TestSubtract` and `TestMultiply` in the same way, to test multiple cases for each function. Try to think of interesting and varied test cases: for example, negative numbers, zeroes, fractional numbers. We're not only testing the relevant functions, we're also documenting their behaviour for anyone who reads or uses the program.

Make sure that when the test fails, the failure message shows exactly which test case didn't work. When you're happy with the expanded tests, move on to the next section.

If you get stuck, have a look at one possible solution in Listing 5.1.

A different kind of problem: division

The board reviewed the calculator project yesterday, and they're really pleased with your progress. The only remaining major feature to be completed is division, and that needs a little more thought than the others.

Why? Because it's possible to write division expressions that have *undefined* results. For example:

```
6 / 0
```

This looks reasonable at first sight, but what we're really asking here is "What number, multiplied by zero, gives 6?" And there is no such number, is there? What should the calculator do in this case?

Error values in Go

There are several options: it could give the answer `0`, which seems unsatisfying, or it could give an answer that represents infinity (perhaps the symbol ∞), or it could define some symbol that means *not a number* (the abbreviation `NaN` is sometimes used for this.)

But these are awkward attempts to work around a fundamental issue: dividing by zero is *not allowed* in the ordinary rules of arithmetic. So there's something about the `Divide` function that wasn't true of `Add`, `Subtract`, and `Multiply`: it's possible to give it *invalid input*. So what value should we return in that case?

A neat thing about functions in Go is that they're not limited to returning just a single value: they can return *multiple* values (and often do). Could this be a solution to the `Divide` problem?

It sounds like `Divide` should return two values. One will be the result of the calculation, as with the other functions, but the other can be an *error* indicator. If something went wrong, like attempting to divide by zero, the error value can convey that information back to the code that called `Divide`.

Go has a special type just for this, called `error`. It's very common for Go functions to return two values, with the first being the result of the computation (the *data value*) and the second being an `error` value. I call this pattern "something and error", where the error value indicates whether or not the data value is valid.

Test behaviours, not functions

How could we *test* such a function? We could certainly test a set of *valid* inputs to `Divide`, in the same way that we did for `Add`, `Subtract`, and `Multiply`. But what about invalid inputs, and what about that extra error result returned from the function?

It's tempting to try to extend our table test to check both the result and the error value at the same time. But that's complicated, and we want tests to be simple.

A useful way to think about this problem is to frame it in terms of behaviours, not functions. What does this mean, exactly?

The *behaviour* of a program describes what it does in different circumstances. For example, the behaviour of the `Add` function is to add its two inputs together and return the result. This is straightforward, and we've already tested that kind of behaviour in a few different cases.

The behaviour of the `Divide` function, given valid input, is to return the computed answer, and a `nil` error value, indicating that everything's fine and there was no error.

But now we're saying there's *another* behaviour, too. If the input is invalid, `Divide`'s error result should be set to something that indicates a problem. For every different thing the function can do according to circumstances, we say it has a distinct behaviour.

A nice rule to remember is:

One behaviour, one test.

This helps us create small, simple, focused tests that are easy to read and understand. Each test should be about one specific behaviour of the program.

So for the "valid input" behaviour, we can use the same basic structure of test that we did for `Add`, `Subtract`, and `Multiply`, with a few small changes. Let's do that, before considering how to test the "invalid input" behaviour.

Testing valid input

Let's start by creating the `TestDivide` function, and copying and pasting the code from one of our existing tests; perhaps `TestAdd`. Here it is:

```
func TestAdd(t *testing.T) {
    t.Parallel()
    type testCase struct {
        a, b float64
        want float64
    }
    testCases := []testCase{
        {a: 2, b: 2, want: 4},
        {a: 1, b: 1, want: 2},
        {a: 5, b: 0, want: 5},
    }
    for _, tc := range testCases {
        got := calculator.Add(tc.a, tc.b)
        if tc.want != got {
            t.Errorf("Add(%f, %f): want %f, got %f",
                tc.a, tc.b, tc.want, got)
        }
    }
}
```

(Listing 5.1)

What needs to change here? Well, we'll be calling the `calculator.Divide` function instead of `calculator.Add`; that's straightforward, so let's make that change. Let's also set up some suitable cases:

```
func TestDivide(t *testing.T) {
    t.Parallel()
    type testCase struct {
        a, b float64
        want float64
    }
    testCases := []testCase{
        {a: 2, b: 2, want: 1},
        {a: -1, b: -1, want: 1},
        {a: 10, b: 2, want: 5},
```

```
    }
    ...
}
```

(Listing 6.1)

Receiving the error value

But there's something else, too. Here's how `TestAdd` sets the `got` value:

```
got := calculator.Add(tc.a, tc.b)
```

We're saying that `Divide` is going to return *two* results, though. The first is `got`, as usual—the answer to the sum. But we're adding a second result: the error indicator. We'll need to receive both these values in the assignment statement. Let's name the second value `err`, short for `error`:

```
got, err := calculator.Divide(tc.a, tc.b)
```

Fine. We know how to compare that `got` with `tc.want` to check the arithmetic; that's no problem. What about the `err` value? What should the test expect it to be?

Well, *this* test is only for the "valid input" behaviour, so we expect there to be no error. We can do this by comparing `err` with the special value `nil` (meaning, in this case, "no error"). If they are different, then the test should fail:

```
if err != nil {
    t.Fatalf("want no error for valid input, got %v", err)
}
```

Why call `t.Fatalf` in this case, rather than `t.Errorf`, as we've been using up to now? What's the difference? Well, `t.Errorf` marks the test as failed, but it still continues executing the rest of the test.

`t.Fatalf`, by contrast, exits the test immediately; it says that things are so broken that there's no point continuing. And this is the case here, since if `err` is not `nil`, that indicates that the `got` value shouldn't be used—it's invalid, because there was an error while trying to produce it.

If we successfully made it through that check without failing, then we know the `got` value is at least *valid*, even if not necessarily *correct*—and we'll check for that now, in the same way as for previous tests:

```
if tc.want != got {
    t.Errorf("Divide(%f, %f): want %f, got %f", tc.a, tc.b,
        tc.want, got)
}
```

And that's all we need to do. Here's the complete body of `TestDivide`:

```
t.Parallel()
type testCase struct {
    a, b float64
    want float64
}
testCases := []testCase{
    {a: 2, b: 2, want: 1},
    {a: -1, b: -1, want: 1},
    {a: 10, b: 2, want: 5},
}
for _, tc := range testCases {
    got, err := calculator.Divide(tc.a, tc.b)
    if err != nil {
        t.Fatalf("want no error for valid input, got %v", err)
    }
    if tc.want != got {
        t.Errorf("Divide(%f, %f): want %f, got %f", tc.a,
            tc.b, tc.want, got)
    }
}
```

(Listing 6.1)

If we run `go test` now, assuming there are no typos, we should expect to see a compile error because `calculator.Divide` isn't defined yet, and so we do:

```
./calculator_test.go:80:15: undefined: calculator.Divide
```

We'll see how to write `Divide` itself in the next chapter.

Takeaways

- When writing several similar tests, don't be afraid to copy and paste an existing test and modify it to do what you want (just don't forget to check that you made all the necessary changes).
- The initial result of writing a test should be a compiler error like `undefined: calculator.Multiply` (because we haven't written that function yet).
- The next step is to write a *null implementation* of the function, that deliberately returns the wrong result, so that we can check that the test detects this situation.
- Once we've verified the test is working correctly, we can go ahead and write the real implementation of the function, and when the test passes, we can be confid-

ent that we've got it right.

- When we want to test some behaviour several times against different inputs, we can use a slice of *test cases* and loop over them, testing each one.
- A `for ... range` loop will execute some bit of code once for each element in the slice.
- There's no value in just testing lots of similar cases: to increase our confidence in the correctness of the function, we should try as many *different* kinds of cases as we can think of: zero, negative, empty, backwards, forwards, sideways.
- Go functions can return more than one result, and often do: the most common pattern is *something and error*, with the error value indicating whether or not the whole operation succeeded.
- If the input to a function can be *invalid*, then an important behaviour of the function is to return a non-`nil` error when this happens: write a separate "invalid input" test for this.
- And in general, use the "one behaviour, one test" rule (rather than, say, "one function, one test", which can lead to some very complicated tests).
- For "something and error" functions, the "error" test only needs to check the error value and make sure it's *not* `nil`: the data value can be ignored, because it's invalid.

3. Errors and expectations

So how do we actually write the `Divide` function in order to pass the test we wrote in the previous chapter? Let's start by copying, pasting, and modifying one of our existing arithmetic functions, such as `Multiply`.

Returning an error value

We might produce something like this as a first draft:

```
func Divide(a, b float64) float64 {
    return a / b
}
```

It looks reasonable, so let's try to run the tests and see what we get.

```
./calculator_test.go:80:12: assignment mismatch: 2 variables
but calculator.Divide returns 1 values
```

Apparently we're not quite there yet. What is this compiler error telling us? Let's first of all look at the relevant file (`calculator_test.go`) and line (80):

```
got, err := calculator.Divide(tc.a, tc.b)
```

The words "assignment mismatch" tell us that the assignment statement here isn't valid because the two sides don't match in some way. We need the same number of variables on the left hand side of the `:=` as there are values to match them on the right. And the compiler is telling us we have "2 variables" on the left, which is perfectly correct, but

the `Divide` function "returns 1 values". This can't be right, because we need 2 values, one for each variable.

Why does the compiler think that `Divide` returns one result value? Because that's what the function signature says:

```
func Divide(a, b float64) float64 {
```

But we've said we now want to return *two* results, of type `float64` and `error` respectively. Let's make that change:

```
func Divide(a, b float64) (float64, error) {
```

Notice that now we have to put parentheses around the list of results: `(float64, error)`. When we had only one result, we could omit those parentheses, but now they're required.

Does this fix the problem? Well, only partially:

```
./calculator.go:30:2: not enough arguments to return
        have (float64)
        want (float64, error)
```

The error messages from the Go compiler can be a bit intimidating when you're not used to them, but they're extremely accurate and comprehensive. They have all the information necessary to fix the problem, if only we can understand them.

Let's break this error message down. First of all, it's saying the problem is in the `calculator.go` file at line 30. Here it is:

```
return a / b
```

And the problem is "not enough arguments to return". In other words, it was expecting more values following the `return` keyword than it actually got. To be even more specific:

```
have (float64)
want (float64, error)
```

So we *have* `float64`; that's the value of `a / b`, which since both `a` and `b` are `float64`, must itself be `float64`. But we *want* `float64, error`. In other words, the function signature promised to return two results, `float64` and `error`, and currently it's only *returning* the `float64` value.

We must supply another value of type `error`. What value could that be? Well, the test is expecting `nil`, as you'll recall. So the minimum we could do here to pass the test is to return `nil` as the second value. Let's make that change:

```
return a / b, nil
```

Now the test passes:

```
PASS
```

```
ok      calculator      0.192s
```

Testing the invalid input case

So we've tested the "valid input" behaviour of `Divide`, with only a few minor adjustments to the test function. What about the "invalid input" case, then?

Let's keep to our plan of "one behaviour, one test". So we'll write a new test function named after the behaviour we're interested in:

```
func TestDivideInvalid(t *testing.T) {
    t.Parallel()
```

Now you take over.

GOAL: Write the body of `TestDivideInvalid`. The test should verify that, given invalid input, `Divide` returns some non-nil error value (but it doesn't matter what it actually is, only that it's not nil.) It doesn't need to check the `float64` result value, because that won't be valid in the error case: we can ignore it.

Here's one version that would work (if yours looks different, that's fine, so long as it behaves the same way):

```
func TestDivideInvalid(t *testing.T) {
    t.Parallel()
    _, err := calculator.Divide(1, 0)
    if err == nil {
        t.Error("want error for invalid input, got nil")
    }
}
```

(Listing 6.1)

Since in this test we're not going to do anything with `got`, we don't need a variable for it. But we have to write *something* on the left-hand side of this assignment, so we use the *blank identifier* (`_`), meaning "I won't need this value".

It seems clear that we'll need to make some changes to `Divide` in order for this test to pass. Let's see what we have right now:

```
func Divide(a, b float64) (float64, error) {
    return a / b, nil
}
```

The second argument to `return` is the error value we're looking for, and currently it's hard-wired to `nil`. In order to get this test passing, we'll first of all need some way to detect if the input is invalid.

Detecting invalid input

We want to do something different *if* the input is invalid, and we already know how to do that in Go: use an `if` statement. For example, given inputs `a` and `b`, we can check if `b` is zero with a statement like this:

```
if b == 0 {
```

It doesn't matter if `a` is zero; the answer will just be zero regardless of what `b` is. But if `b` is zero, that's an impossible division, so that signals our error behaviour.

Now we're ready to return something different in this case. For the `float64` result (the data value), it really doesn't matter what we return. The semantics of "something and error" functions say that a non-nil error means the other result is invalid and should be ignored. The convention is to return zero, so we can do that here too.

What about the error value? We know `nil` won't do here, because the test requires it to be something else. But what other values can errors have, and how do we construct them?

Constructing error values

The standard library `errors.New` function is designed for exactly this. It takes a single string argument, and returns a value of type `error`:

```
return 0, errors.New("division by zero not allowed")
```

If we print that value using `fmt.Printf` (or `t.Errorf`), it will print the string message, as we'd expect. So here's our modified `Divide` function:

```
func Divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division by zero not allowed")
    }
    return a / b, nil
}
```

(Listing 6.1)

This looks pretty simple, but it does everything we need:

```
PASS
ok      calculator      0.185s
```

Do we need more cases? Not really. The only invalid input is zero, and we've tested that. In general, tests for the "error expected" behaviour can be quite short and simple, which is nice. But if we're writing a function that can return an error for various different kinds of invalid input, we should use a table test with multiple cases.

For now, though, we're all done with `Divide`. Nice work! Let's review what we've learned so far.

The structure of tests

All tests follow more or less the same basic structure:

- Set up the inputs for the test, and the expected results
- Call the function under test and pass it the appropriate input
- Check that the result matches the expectation, and if it doesn't, fail the test with a message explaining what happened

And we've also practiced a specific workflow, or sequence of actions, that we can use when developing any piece of Go code:

The development process

1. Start by writing a test for the function (even though it doesn't exist yet)
2. See the test fail with a compilation error because the function doesn't exist yet.
3. Write the minimum code necessary to make the test compile and fail.
4. Make sure that the test fails for the reason we expected, and check that the failure message is accurate and informative.
5. Write the minimum code necessary to make the test pass.
6. Optionally, tweak and improve the code, preferably without breaking the test.
7. Commit!

This is sometimes referred to as the "red, green, refactor" cycle. The "failing test" stage is "red", the "passing test" stage is "green", and beautifying the code is "refactor". In practice, we may go through this loop a few times with any particular test or function.

Any function with reasonably complicated behaviour is probably too hard to test and implement from scratch, so try to break its behaviour down into smaller chunks. These might actually be separate functions, or maybe just sub-behaviours that we can write individual tests for. Once we have most of the parts we need, we can use them to build up the real function, step by step.

Comparing floating-point values

The way of representing decimal fractional numbers that's used in most programming languages is called *floating-point* (that's why Go's fractional data type is called `float64`: it's a 64-bit floating-point number).

And one thing about floating-point numbers is that they have limited *precision*. If you think about the decimal representation of a fraction like one-third, it's `0.33333333...`, but it never stops. It's threes all the way down. Representing a

number like this in a fixed number of bits (or even on a finite-sized piece of paper) is bound to involve some loss of precision.

Let's try adding a case like this to our existing `TestDivide` cases:

```
{a: 1, b: 3, want: 0.333333,},
```

This fails with the rather puzzling result:

```
Divide(1.000000, 3.000000): want 0.333333, got 0.333333
```

It looks as though `want` and `got` are the same, yet the test is failing. The difference between them is tiny, less than we can see when printing out the values, but enough to make them not equal according to the == operator. So that's not the right way to compare such values.

Instead, what we can do is add a function to do a looser comparison: not *equal*, exactly, but just "close enough", for some value of "enough".

Let's put this in the `calculator_test` package, which is where we'll need to use it:

```
func closeEnough(a, b, tolerance float64) bool {
    return math.Abs(a-b) <= tolerance
}
```

Then we can compare our `tc.want` and `got` values using this function instead of ==:

```
if !closeEnough(tc.want, got, 0.001) {
```

The third argument specifies how far apart the values are allowed to be and still be considered "equal" for our purposes. In this case, that gap is 0.001. We may need to tune this value depending on the test cases, or the requirements for the specific program we're working on. For example, a navigation program for a spaceship may require fewer decimal places of accuracy than a robotic brain surgery system.

A `Sqrt` function

Our basic calculator is complete, but we've just received an urgent message from the VP of Sales. She wants an extra premium feature that can be used to upsell users to the *Enterprise* calculator. The ordinary calculator has been re-branded as the *Home* edition.

It seems that enterprise users would like the ability to take square roots, so the VP's asking us to develop a `Sqrt` function. We already know everything we need to know to design, test, and develop this feature, so let's get started!

GOAL: Develop a `Sqrt` function that takes a `float64` value and returns its square root as a `float64` value.

Use any standard or third-party packages you think are appropriate. The function should return an error for negative inputs (no real number squared equals a negative

number, so taking the square root of a negative number is not a valid operation). The test should check the error handling in the same way as the tests for `Divide`.

Don't forget, because of the inexactitude of floating-point calculations, we may need to use a function like `closeEnough` to compare the results with expectations. Depending on the test cases, we may need to make the tolerance value larger; perhaps as large as 0.1.

When the feature is complete, you're done. Nice work. The company has awarded you a sizable bonus, and an all-expenses-paid vacation. Enjoy it!

If you get stuck, or if you'd like to compare your solution with mine, have a look at the sample implementations of `TestSqrt`, `TestSqrtInvalid`, and `Sqrt` in Listing 8.1.

Running programs

So far in this book, we've written tests and functions in Go, and run the tests using the `go test` command. That's all very well, but you might be wondering how to run a *program* in Go, rather than just a set of tests. In this section, we'll see how to do exactly that.

In order for any program to run on a computer it needs to be in the form of an *executable binary*: a file containing machine code that the computer's CPUs can understand, in the particular format required by its operating system.

To turn a text file containing source code into an executable binary file, Go uses a program called a *compiler* (that's what gives us all the angry messages about undeclared variables and mismatched types). The compiler translates Go programs into machine code, producing an executable binary that we can then run.

So how does that work?

The `main` package

In order for a bunch of Go source code to produce an executable, there has to be at least some code in a special package named `main`. Since we can't have more than one package in the same folder (apart from test packages), we'll need to put the `main` package in a subfolder of your project.

Because the `main` package produces a *command* (that is, an executable), it's conventional to name this folder `cmd` (short for "command").

It's not enough just to declare a `main` package, though. Whatever our program does, it has to *start* somewhere, and the Go compiler needs to know where that should be. The way we tell it is by defining a special function, also named `main`.

The flow of control in a Go program begins at the beginning of the `main` function, and works through it in statement order, perhaps calling other functions along the way.

When (and if) execution reaches the end of `main`, the program stops.

The `go run` command

Let's write an executable program that has a `main` package and a `main` function, then. We'll use our `calculator` package to perform some calculation, and print the result in the user's terminal.

Create a new subfolder named `cmd` in the `calculator` project. Because there could be more than one command associated with this package, let's create a further subfolder `calculator`, too:

```
mkdir -p cmd/calculator
```

Add a file named `main.go` in this folder, containing the following:

```
package main

import (
    "calculator"
    "fmt"
)

func main() {
    result := calculator.Add(2, 2)
    fmt.Println(result)
}
```

(Listing 8.2)

We know that the `package main` declaration is required if we want to produce an executable binary. And we also need a function `main`, or the binary wouldn't do anything.

Here's the folder structure we should end up with:

```
calculator/
    calculator.go
    calculator_test.go
    cmd/
        calculator/
            main.go
    go.mod
```

We can run this program directly from the command line by using the `go run` command:

```
go run cmd/calculator/main.go
```

Here's the result:

```
4
```

That's encouraging!

The `go build` command

The `go run` command is useful for quickly checking whether the program works, but if we actually want to *distribute* our software to customers, a more convenient way to do that is to build an executable file. Let's see how that works.

To create the executable, run the command:

```
go build -o add ./cmd/calculator
```

Windows works slightly differently, so if you're on that platform you'll need to run instead:

```
go build -o add.exe ./cmd/calculator
```

If there are no errors, this command won't produce any output, but you'll find that a new file has been created in the current directory, named `add`.

To run it, use the program name itself as the command:

```
./add
```

```
4
```

On Windows, run just:

```
add
```

Great job! We'll have more to say about `go build` and what it can do later in the book, in the "Building blocks" chapter, but this is all we need for the moment.

Now that you've successfully developed and shipped the calculator application, your engagement at Texio Instronics is complete. It's time to move on to your next gig: helping to build an online bookstore.

Takeaways

- A common compiler error is something like `assignment mismatch: 2 variables but X returns 1 values`: this means that we tried to receive a different number of results than the function actually returns.
- When a function declares multiple result values, the list of result types must be enclosed in parentheses: for example, `(float64, error)`.

- Another common kind of compiler error is `not enough arguments to return`, meaning that the function *declares* a certain number of result values, but there's a `return` statement with a different number of arguments.
- When there's no error, return the explicit value `nil` as the `error` result: `nil` means "no error".
- The compiler won't let us declare a variable that we never refer to again (it correctly infers that this must be a mistake).
- But sometimes we *need* a variable syntactically, as in the left-hand side of an assignment statement: in this case, we can use the *blank identifier* `_` to act as a placeholder.
- The test failure for "invalid input" cases should say something like "want error for invalid input, got nil", and since there's no data in this message, we use `t.Error` instead of `t.Errorf`.
- When a function detects an error, it should return zero for the data value, or whatever the appropriate zero value is for its type, along with the error.
- `if` statements often use `==` or `!=` to compare two values, but there are other kinds of comparison *operators*: for example, `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to).
- We can also join together two or more expressions using *logical operators* like `&&` (true if all expressions are true) and `||` (true if at least one expression is true).
- To *negate* an expression, use the "not" operator `!` (true if the expression is false).
- People make a lot of fuss about testing, but it's essentially very simple: first we work out what we `want`, and then we compare it against what we actually `got`.
- Our workflow can be summarised by the phrase "red, green, refactor": write a failing test (red), implement the code to make it pass (green), and then refactor both the code and the test.
- If you can't figure out how to test something, try testing a smaller, simpler function with less behaviour; similarly, if you can't figure out how to implement something, use a simpler function as a stepping-stone.
- Tests are all about comparisons, but we need to be careful when comparing floating-point values, because there's an inherent loss of precision in the way computers store these numbers.
- An exact comparison may not work with some values, in which case we can instead check whether the `want` and `got` are *close enough*, for some definition of "enough".
- The `go run` command both compiles *and* executes a program, which is useful for testing.

4. Happy Fun Books

Welcome aboard! It's your first day as a Go developer at **Happy Fun Books**, a publisher and distributor of books for people who like cheerful reading in gloomy times. You'll be helping to build our new online bookstore using Go.

First, let's talk a little about *data* and *data types*. By "data", we really just mean some piece of information, like the title of a book, or the address of a customer. So can we distinguish usefully between different *kinds* of data?

Yes. For example, some data is in the form of text (like a book title). We call this *string* data, meaning a sequence of things, like "a string of pearls", only in this case it's a string of characters.

Then there's data expressed in numbers, like the price of a book; let's call it *numeric* data.

There's also a third kind of data we've come across: *boolean* data that consists of "yes or no", "true or false", "on or off" information.

Types

The word "type" in programming means what you think it means, but with a special emphasis. It means "kind of thing", but in a way that allows us to automatically make sure that our programs are working with the right kind of thing.

For example, if we declare that a certain variable will only hold string data (we say the variable's *type* is `string`), then the Go compiler can check that we only assign values of that type to it. Trying to assign some other type of value would cause a compile error.

Let's distinguish first between a *value* (a piece of data, for example the string `Hello, world`), and a *variable* (a name identifying a particular place in the computer's memory that can store some specific value).

Values have types; the type of `Hello, world` is named `string`. So we call this a *string value*, because it's of *type* `string`. A variable in Go also has a type, which determines what values it can hold. If we intend it to store string values, we will give it type `string`, and we refer to it as a *string variable*.

So a string variable can hold only string values. That's straightforward, isn't it? Make sure you feel comfortable in your understanding of types, values, and variables before continuing.

Variables and values

How do we create and use variables and values? Let's find out. Go ahead and create a new folder, copy the example below to a file called `main.go` and run it with the command `go run main.go`.

```
package main

import "fmt"

func main() {
    var title string
    var copies int
    title = "For the Love of Go"
    copies = 99
    fmt.Println(title)
    fmt.Println(copies)
}
```

(Listing 9.1)

What's happening here? Let's break it down. As you learned in the previous chapter, all executable Go programs must have a `package main`. They also have a *function* called `main` that is called automatically when the program runs. Let's see what happens inside `main`.

First, we *declare* a variable `title`, of type `string`:

```
var title string
```

(Listing 9.1)

The `var` keyword means "Please create a variable called... of type...". In this case, please create a variable called `title` of type `string`.

Next, we declare a variable `copies` of type `int` (a numeric type for storing integers, that is, whole numbers).

```
var copies int
```

(Listing 9.1)

Having created our variables, we're ready to assign some values to them. First, we assign the string value `"For the Love of Go"` to the `title` variable, using the = operator.

```
title = "For the Love of Go"
```

(Listing 9.1)

The variable name goes on the left-hand side of the =, and the value goes on the right-hand side. You can read this as "The variable `title` is assigned the value `"For the Love of Go"`".

This value, you now know, is a string *literal* ("It's literally this string!").

Next we assign the value `99` (another literal) to the `copies` variable, in the same way. Good job there's no shortage of copies of this important book!

```
copies = 99
```

(Listing 9.1)

Finally, we use the `fmt.Println` function to print out the values of `title` and `copies`:

```
fmt.Println(title)
fmt.Println(copies)
```

(Listing 9.1)

If you run this program, what output will you see? Try to work it out before running the program to check if you were right.

GOAL: Write a Go program that declares a variable of a suitable type to hold the name of a book's author. Assign it the name of your favourite author. Print out the value of the variable. Run your program and make sure it's correct.

Type checking

We said that the Go compiler will use the type information you've provided to check whether you accidentally assigned a value to the wrong type of variable. What does that look like? Let's try something silly:

```
title = copies
```

We know straight away that this won't work, right? `title` is a string variable, so it can only hold string values. But `copies` is an `int` variable, so we can't assign its value to anything but an `int` variable. If you add this line to the program, what happens? We get an error message from the compiler:

```
cannot use copies (type int) as type string in assignment
```

That makes total sense, given what we know. Another way to phrase this would be "You tried to assign a value of type `int` to a `string` variable, and that's not allowed".

Type checking happens in other places too, such as when passing parameters to a function. Suppose we declare a function `printTitle`, that takes a string parameter:

```
func printTitle(title string) {
    fmt.Println("Title: ", title)
}
```

When we *call* this function, we need to *pass* it a value of the specified type. So `title` would be okay here, because it's of type `string`, and so is the function's parameter:

```
printTitle(title)
```

You already know that trying to pass it an `int` value won't work, but let's try it:

```
printTitle(copies)
```

Just as we thought, that's a *type error*:

```
cannot use copies (type int) as type string in argument to
printTitle
```

Type errors of this kind can happen, especially in more complicated programs. Fortunately, Go is on our side, and will tell us right away if we get our types in a twist.

GOAL: Add a variable to your program to represent the current edition number of the book: first edition, second edition, and so on. Assign it a suitable value.

More types

Besides the types `string` and `int`, which are probably the most commonly used in Go, there's also `bool`, for boolean values, such as `true` or `false`, and `float64` for decimal numbers with fractional parts, such as `1.2` (don't worry about the weird name `float64` for now; just read it as "fraction"). Here are some examples:

```
var inStock bool
inStock = true
var royaltyPercentage float64
royaltyPercentage = 12.5
```

GOAL: Declare a variable of a suitable type to represent whether or not the book is on special offer, and assign it a value. Do the same with a variable that stores a discount percentage (for example, 10% off).

Zero values and default values

You might have wondered what happens if you declare a new variable using the `var` keyword, but then don't assign it any value. What happens when you try to print the variable, for example? Let's see:

```
var x int
fmt.Println(x)
// 0
```

How interesting! It turns out the variable has a *default value*, which is what it contains before we explicitly put anything in it. What that value actually *is* depends on the type.

Each type in Go has a *zero value*, which is the default value for variables of that type. For `int` (and `float64`, or any numeric type), the zero value is zero, which makes sense, but what about other types?

The zero value for `string` is the empty string, `""`, which also seems logical. Speaking of logic, what about `bool` variables? They default to `false`.

The fact that variables declared with `var` get the zero value by default leads to an interesting point of Go style. When we declare a variable and we would *like* its starting value to be zero, then we declare it with `var`:

```
var x int
```

On the other hand, when we want the starting value of `x` to be something other than zero, we tend to use the short declaration syntax (`:=`) to both declare and assign it in one statement:

```
x := 1
```

If you've ever wondered when to use `var` and when to use `:=`, then this is one way to decide. If you *want* the default value, use `var`. If you want some other value, assign it with `:=`.

Introducing structs

Here at Happy Fun Books, we deal in many different kinds of data. Books are one example: a book has various kinds of information associated with it, including the title, author, price, ISBN code, and so on. In our online store, we'll also need to deal with data about customers (name, address, and so on), and orders (which customers have ordered which books, for example).

Instead of dealing with each piece of book data separately, as we've done so far, we'd prefer to deal with a book as a *single* piece of data in the system, so that we can (for example) pass a book value to a function, or store it in some kind of database.

A data type made up of several values that you can treat as a single unit is called a *composite* type (as in, "composed of several parts"). One of the most useful composite data types in Go is called a *struct*, short for "structured record".

A struct groups together related pieces of data, called *fields*. For example, a struct describing a customer of our bookstore might have a field called `Name` for their name (a string would make sense for this). Another field we might need is `Email`, for their email address (let's use `string` for this too).

Here's what a definition of a simple `Customer` type might look like, then:

```
// Customer represents information about a customer.
type Customer struct {
    Name  string
    Email string
}
```

There's more information we could store about customers, but this would be enough to get us started.

Recall that earlier in this book you used a `testCase` struct type in your tests for the `calculator` package, to define the inputs and expected results for each case. Structs are a very convenient way to create a single value that contains multiple pieces of data inside it.

Type definitions

We begin with a comment, introduced by the `//` characters. It's good practice to add an explanatory comment for each type you create in Go, telling readers what the type is intended to do. These are called *documentation comments*, because when you publish your project to a hosting site such as GitHub, your comments will be automatically turned into browsable documentation on the pkg.go.dev site. You can see some examples here:

https://pkg.go.dev/github.com/bitfield/script

Next, we can see a new keyword here: `type`. This introduces a new *type definition*. It's followed by the name we want to give the type (`Customer`). It's a bit like how the `var` keyword declares a variable. With the `type` keyword, we're saying "Please create a type called... and here are the details..."

The details, in this case, are the keyword `struct` (which introduces a struct type), and a list of fields inside curly braces `{ ... }`. Everything after `type Customer ...` is

called a *type literal*. We've seen values before that were string literals, for example; here we have a type literal.

Exported identifiers

You might have wondered why the type is called `Customer`, with a capital C, instead of just `customer`. It turns out that names with initial capital letters have a special meaning in Go. Anything with such a name is available outside the package where it's defined (we say it's *exported*). If you define a type (or a function, a variable, or anything else) whose name starts with a lowercase letter (`customer`, for example), it is *unexported* and so you won't be able to refer to it in code that's outside this package (in a test, for example).

You could think of exported names, with the initial capital letter, as identifying things that the package intends to be *public*, while the unexported names, with a lowercase initial letter, are *private* things that the package is going to use internally, but no one else needs to know about.

The core package

As you know, the *package* is Go's way of organising chunks of related code. With most projects, there's some kind of *core* package that implements the most basic functionality relating to the problem we're solving. What would that be for our bookstore, and what should we name it?

A single word is best, ideally a short word, and one that completely describes what the software is for. Let's call our core package `bookstore`.

The very first test

As you'll recall from our earlier work on the `calculator` package, we need to first of all decide what *behaviour* we want, and then write a *test* that specifies that behaviour precisely.

Only once we're happy with the test, and we've seen it fail when we *know* the system is incorrect, should we go on to write the code that *implements* that behaviour.

So what's the first behaviour we should try to describe and test for our bookstore application? We know we want a core `Book` type in our `bookstore` package, and it's probably a good idea to set this up before anything else, but how can we create one if we're not allowed to write any non-test code yet? What's the behaviour that we would express as a test that requires `Book` to be implemented?

Let's get the project structure in place before answering this question. Create a new folder for the `bookstore` project if you haven't yet, and run `go mod init bookstore` in it to create a new Go module here.

Create a file named `bookstore_test.go`, and let's start with the usual test preamble:

```
package bookstore_test

import (
    "bookstore"
    "testing"
)
```

(Listing 10.1)

Now we're ready to think about the first test.

Testing the core struct type

A struct type doesn't really have any behaviour by itself, but we'd like to at least establish the core `Book` type before we move on. So let's write a very simple test that doesn't actually *do* anything, but can't pass unless the `Book` type has been defined.

That would mean that we need to use a value of that type. What would that look like?

Struct literals

We've seen some literals of basic types already (`99` is an `int` literal, for example). Can we write struct literals too? Yes, we can. They look like this:

```
bookstore.Book{
    Title:  "Nicholas Chuckleby",
    Author: "Charles Dickens",
    Copies: 8,
}
```

The literal begins with the name of the type (`Book`), followed by curly braces. Then we write inside the curly braces a list of pairs of field names and values (for example, `Title: "Nicholas Chuckleby"`). Each field, including the last one, must be followed by a comma. Note that the struct definition itself doesn't have commas at the end of each line, but struct literals must.

Assigning a struct literal

So now we know how to write a value of type `Book`, what shall we do with it? Here's one idea:

```
func TestBook(t *testing.T) {
    t.Parallel()
    _ = bookstore.Book{
        Title:  "Spark Joy",
        Author: "Marie Kondo",
        Copies: 2,
    }
}
```

(Listing 10.1)

Wait, what? Is that a test? *How even?*

It seems like this isn't really testing anything, because there's no way for it to fail. There's no call to `t.Error`, for example, or even any `if` statement that could *decide* whether the test passes or fails. It should always pass.

So what's actually happening in this test? There's some struct literal of type `Book`, and we're assigning it to something. To what?

Remember earlier when we used the blank identifier (`_`) to ignore a value, because we didn't need it? It's the same thing here. We don't want to *do* anything with this `Book` value. We just want to *mention* it, to help us get our ideas straight about what the `Book` type is and what fields it has.

For that, we need a literal, and since we can't just write a literal on its own (that's not a valid Go statement) we need to assign it to something. If we assigned it to some named variable, though, the compiler would complain that we didn't then use that variable, so the blank identifier is our excuse here.

The unfailable test

A test like this can't fail, so it should always pass. But *does* it actually pass right now? No: not because it fails, but because it doesn't compile. And that makes sense, because it imports a non-existent package `bookstore`, and refers to a data type in it (`bookstore.Book`) that doesn't exist either.

There's no way this test can compile correctly until those things are defined somewhere (we could call it a *compile-only test*). Before defining them, we would like to be at the point where this test fails to compile with some message like `undefined: bookstore.Book`.

Let's create a new file `bookstore.go` and add a package declaration:

```
package bookstore
```

(Listing 10.1)

If we run `go test` now, we can see that we're at the point where we wanted to be:

```
# bookstore_test [bookstore.test]
./bookstore_test.go:10:6: undefined: bookstore.Book
FAIL    bookstore [build failed]
```

We knew that the test couldn't compile unless the `Book` type was defined and had exactly the fields that the literal value in the test requires, and it isn't and doesn't... yet. But we're very close. Over to you!

GOAL Make this test pass. Hint: you'll need to define a type in package `bookstore` named `Book`. The compiler will tell you if you haven't defined it quite right. Just keep tweaking it and re-running the test until you've got the struct definition the way it needs to be.

Don't forget, because we're referring to the `Book` type from *outside* the `bookstore` package, we need it to start with a capital letter. If you defined this type with the name `book`, it would be unexported and therefore you wouldn't be able to use it in the test.

Here's my version. If yours doesn't look exactly the same, that's okay, as long as it passes the test. The test is the *definition* of what's okay!

```
// Book represents information about a book.
type Book struct {
    Title  string
    Author string
    Copies int
}
```

(Listing 10.1)

Takeaways

- Go has various *data types* for dealing with different kinds of information: `string` holds text, `int` holds integer numbers, `float64` holds fractional numbers, and `bool` holds Boolean (true or false) values.
- The Go compiler checks that a variable or parameter of a given type is only ever assigned values of that type: if it detects a mismatch, that's a compile error.
- The `var` keyword introduces a variable, and specifies its type, so that we can use it later on in the program.
- A *literal* is a value of some type that we specify directly in our Go code, such as the string literal `"For the Love of Go"`.
- Variables in Go have a *default* value before we assign something to them explicitly: it's whatever the *zero* value is for its type, such as `""` for strings, 0 for numbers, or `false` for `bool`.

- We often declare and assign values to variables in a single statement, using the *short declaration form*, for example `x := y`.
- When you want to group related values together into a single variable, you can define a *struct type* to represent them.
- The different *fields* of the struct are just like any other kind of Go variable, and they can be of any Go type, including other structs.
- Type definitions are introduced by the keyword `type`, and struct types with the additional keyword `struct`, followed by a list of fields with their types.
- Identifiers (that is, the names of functions, variables, or fields) with an initial capital letter are *exported*: that is, they're public and can be accessed from outside the package where they're defined.
- On the other hand, identifiers beginning with a lowercase letter are *unexported*: they're private to the package.
- Most Go packages are "about" one or more core struct types, in some sense: they're often the first things that you'll define and test.
- A neat way to design a struct type guided by tests is to write a *compile-only test*, that simply creates a literal value of the struct: this test can't pass (or, indeed, compile) until you've correctly defined the struct type.

5. Story time

You've made a great start on the Happy Fun Books project, by setting up your basic package structure and core `Book` type. When we get to a point like this in a software project, it's not always easy to know what to do next, or even more importantly, what to do *first*.

User stories

A great way to approach this is to think about it from the point of view of *users* of our software. Even if this is only a fun project for recreation or learning, and we don't anticipate having any *real* users other than ourselves, we can use it as a thinking tool, and imagine what users would want to do with the software and how they'd like to interact with it.

An online bookstore, for example, will need to have certain features in order to be usable for its customers. People will need to see what books are available, see the details of any given book, and perhaps search for a book they're interested in. No doubt you can think of many more such interactions.

We sometimes call these *user stories*, because each of them tells a little story about a user and something they want to do. Sometimes people talk about *features* instead, but this doesn't necessarily make clear that all the work we do on the program should be aimed at interactions with users in some way. And there are lots of features a program could have that are in fact of no interest or value to users. (That doesn't always stop

software vendors from adding them, though.). As developers, starting with a user story helps keep our minds firmly focused on the user's perspective.

What are the core stories?

Now, there are lots of possible user stories for an online bookstore. Here's one: "As a customer, I want to see a sample preview of the book in my browser so that I can figure out if I'd like to read it." Or "As a bookstore owner, I want to see my total revenue for the month, so that I'll know if I can afford to take my family out for pizza yet." I'm sure you can think of dozens more.

So we don't start by listing all possible user stories we *could* implement. That would be a very long list, and we'd have no idea what is the most important story, or which one we should implement first.

Instead, it's a good idea to ask what user stories we couldn't possibly *leave out*. In other words, for the bookstore to be usable at all, there are a few non-negotiable things users need to be able to do. Here are three I can think of:

1. Buy a book
2. List all available books
3. See details of a book

Identifying these *core stories* is a very helpful way to think about any product. Assuming that it costs us money (or at least time) to develop, and it can't start *making* us money until it's viable, it makes sense to focus on developing only the critical stories first: the ones without which we wouldn't even really *have* a product, at least not one that users would want.

The first story

Let's start with the "buy a book" core story, and plan how to implement it. As you'll recall from our work on the `calculator` package, I find it very useful to think in terms of *behaviours*, rather than, for example, functions. Again, this keeps our minds focused on users and what they want, instead of what kind of program architecture seems logical to a software engineer.

What's the behaviour we'll need for the "buy a book" story? Let's simplify it by not worrying about payments for the time being, and just look at it from the point of view of stock control. When a user buys a book, that will necessarily decrease the number of copies available.

That's a good enough verbal description of the behaviour that we can start sketching out a test that expresses the same idea in Go code. In order to write that test, we'll be using the `Book` struct again, and this time we'll need to do some work with variables. Let's take a quick tour of the necessary syntax first.

Struct variables

You're already familiar with using the `var` keyword to declare variables of the built-in types such as `string`. You'll be pleased to know that we can declare variables of a struct type in just the same way:

```
var b bookstore.Book
```

Just as before, we give the keyword `var` followed by the name of the variable, and the type it should be. In our test code, we'll be outside the `bookstore` package, so we need to use the fully-qualified name of this type: `bookstore.Book`.

The short declaration form

You already know that you can declare a variable and assign it a value in one go, using a `var` statement with `=`:

```
var b = bookstore.Book{
    Title:  "Nicholas Chuckleby",
    Author: "Charles Dickens",
    Copies: 8,
}
```

And you may also recall from the `calculator` example that there's a *short declaration form* that uses `:=`, like this:

```
b := bookstore.Book{
    Title:  "Nicholas Chuckleby",
    Author: "Charles Dickens",
    Copies: 8,
}
```

So we can create some variable `b` that holds a `Book` value. What can we do with it? How can we access those fields like `Title` and `Author`?

Referencing struct fields

Recall from your `calculator` tests that to refer to one of the fields on the `tc` variable representing the test case, such as `want`, we used this *dot notation*:

```
tc.want
```

When we come to write our test for the "buying a book reduces the number of copies available" behaviour, we'll need to refer to fields on our `Book` struct too. For example, if

we have some `b` variable representing a book, we could get the value of its `Copies` field using the dot notation:

```
b.Copies
```

Writing the test

You now know everything you need to write the test for the "buy a book" story. It will follow the same basic structure that we used for the `calculator` tests: set up your inputs and expectations (your `want`), call a function, and check the results (your `got`).

What function could we call? Let's give it the name `Buy`, for now (we can always change this later). What does it take? Some `Book` value would make sense. What does it return? Perhaps the modified `Book` value. What does it do? Well, all we need it to do right now is decrease the number of copies of the book by 1.

Suppose we did something like this: call the `Buy` function with some book that has a non-zero number of copies, and then check the result to make sure it has one *less* copy than we started with.

GOAL: Write `TestBuy` along these lines. Don't worry about making it pass yet; just write the complete test function that expresses the logic we just outlined. It should fail to compile because the function you're calling isn't defined; that's okay.

How did you get on? Here's a version that should do the job:

```
func TestBuy(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title:  "Spark Joy",
        Author: "Marie Kondo",
        Copies: 2,
    }
    want := 1
    result := bookstore.Buy(b)
    got := result.Copies
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing 12.1)

Based on our plan, we first of all set up our `b` and `want` variables, then call the `Buy` function, and check that we `got` the expected number of copies remaining.

If not, we fail the test with an informative error message. It's not enough just to say `failed`; what failed? We can't just say `unexpected number of copies`, either; in order to fix the problem, we'd need to know what the *actual* number of copies was. It's not even really sufficient to say `want X copies, got Y`; we need to make it clear that if we started with 2 copies and bought one, we should have one left, but we *got* some other number.

Maybe this seems excessive, but you'll appreciate this level of detail one day when you suddenly get a failing test for no apparent reason and everyone around you is giving you meaningful looks, wondering why they can't ship. When tests fail, they should immediately tell you what to fix, in the clearest and most explicit way possible.

Getting to a failing test

At this point we expect to see a compile error because the `Buy` function isn't yet defined, and that's the case:

```
# bookstore_test [bookstore.test]
./bookstore_test.go:23:12: undefined: bookstore.Buy
```

Let's think about how to implement `Buy` now. As you'll recall, the next step is to write the minimum code necessary to make the test compile, and fail. Over to you.

GOAL: Write the minimum code necessary to make this test compile and fail.

The function signature is already determined by the test, so that's easy:

```
func Buy(b Book) Book {
```

But what to return? We usually start by returning the zero value of whatever the result type is; in this case, `Book`. The zero value of any struct type is just an empty struct literal of that type:

```
return Book{}
```

That should be enough to get the test failing:

```
bookstore_test.go:26: want 1 copies after buying 1 copy
from a stock of 2, got 0
```

Before we can actually make this test pass, we'll need to know a couple more bits of Go syntax.

Assigning to struct fields

Suppose that inside the `Buy` function, we have our `b` parameter, which we know is a `Book`, and we need to update it to reflect the fact that there is one less copy left in stock. How can we do that?

First of all, can we assign some value directly to the `Copies` field? Yes: using the dot notation, we can assign to `b.Copies` just as though it were an ordinary variable. For example, to assign it the literal value 7:

```
b.Copies = 7
```

But we don't want to assign it the value 7; that won't work here. Instead, we need to assign it a value that is *one less* than the current number of copies. That's straightforward:

```
b.Copies--
```

This is called a *decrement* statement; the `--` decrements the value of the field by 1. If, on the other hand, we'd wanted to *increment* the field, we could have written:

```
b.Copies++
```

The `++` statement increases the variable's value by 1. Incidentally, if we want to adjust some variable's value by *more* than 1, we can use this form:

```
b.Copies -= 5
```

Can you guess what this does? That's right: it subtracts 5 from the value of `b.Copies`.

Similarly, `b.Copies += 10` would increase `b.Copies` by 10.

Implementing Buy

You now have all the syntax you need to write the real logic for the `Buy` function. Over to you!

GOAL: Implement the `Buy` function, so that it passes the test.

Here's my version:

```
func Buy(b Book) Book {
    b.Copies--
    return b
}
```

(Listing 12.1)

You'll notice that the function itself is much shorter and simpler than the test that tests it. That's quite normal. After all, a test needs to *describe* the entire behaviour required, whereas the function only has to *implement* it. If about half your codebase consists of tests, overall, that's probably about right. More is fine. Less might indicate that you're missing a few tests.

Test coverage

While we're on the subject, let's look a little more closely at the relationship between test code and the code it tests (often called *system* code to distinguish it from test code).

Throughout this book, we've designed functions by starting with a user story, working out the behaviours needed to implement it, expressing them in code as tests, and then writing the minimum code necessary to get the test passing. So, every line of system code should, in theory, be executed (we say *covered*) by some test. But is that the case?

The Go tools provide a helpful code coverage function to check this. For example, if you run `go test -cover`, you'll see the result:

```
PASS
coverage: 100.0% of statements
```

That's great, but what if not all statements *were* covered by tests? How could we see what the uncovered ones were? Luckily, Go can generate a *coverage profile* for us:

```
go test -coverprofile=coverage.out
```

We can now use this profile to generate an HTML page showing the covered code highlighted in green, uncovered in red, and ignored (things that aren't statements) in grey:

```
go tool cover -html=coverage.out
```

This will open your default web browser to view the generated HTML.

Visual Studio Code's Go extension lets you highlight covered code right there in the editor, using the command *Go: Toggle Test Coverage in Current Package*. Other editors have similar facilities.

You'll notice that some lines are not coloured at all. These are typically things like type definitions, constants, and so on. Although these are part of the program, they're really instructions to the compiler, and don't result in any generated object code in your program binary, so the coverage tool ignores them. They're not counted towards the coverage statistics.

As we've seen, if you're developing your program guided by tests from the start, there shouldn't really be any lines of system code that aren't covered by tests. But what would it look like if there were? Let's try an experiment.

Test-last development

Suppose you're assigned to the Happy Fun Books project to implement a requested feature, and you're not very experienced with writing tests, so you just charge right in and add a piece of system code. This may be covered by the existing tests, but most likely it won't, so we should be able to use the coverage tools to detect the uncovered statements.

It seems that some books are so popular that they've been selling out completely, but right now, the system doesn't detect this situation. If there are zero copies of *Spark Joy* left in stock, the Buy function will happily decrement this value and return the book with its `Copies` set to -1.

That makes no sense, and some angry customers have complained that they've received a negative number of books, and therefore we owe them money. Bad times. Let's fix that.

The basic problem here is that `Buy` doesn't check whether `Copies` is zero before decrementing it. So we can add that check, but what should happen if it is zero? We should probably return some error:

```
if b.Copies == 0 {
    return Book{}, errors.New("no copies left")
}
```

Recall that in the error case, we conventionally return zero for the data value, since it's to be ignored (in this case, that would be `Book{}`).

This doesn't fit with the declared signature of `Buy`, as currently it's only supposed to return one value. We'll need to adjust that to match:

```
func Buy(b Book) (Book, error) {
```

And *that* breaks the original `return` statement, so we can fix that too:

```
return b, nil
```

Here's the resulting Buy function:

```
func Buy(b Book) (Book, error) {
    if b.Copies == 0 {
        return Book{}, errors.New("no copies left")
    }
    b.Copies--
    return b, nil
}
```

(Listing 13.1)

That looks about right by inspection, so let's run the test again:

```
# bookstore_test [bookstore.test]
./bookstore_test.go:25:9: assignment mismatch: 1 variable
but bookstore.Buy returns 2 values
```

Again, that's expected: we changed the signature of `Buy` to return two values, so we need to update the test to receive both of them:

```
result, err := bookstore.Buy(b)
```

The compiler rightly reminds us that we need to do something with `err`, and the natural thing to do is check it:

```
if err != nil {
    t.Fatal(err)
}
```

So here's the modified test in full:

```
func TestBuy(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title:  "Spark Joy",
        Author: "Marie Kondo",
        Copies: 2,
    }
    want := 1
    result, err := bookstore.Buy(b)
    if err != nil {
        t.Fatal(err)
    }
    got := result.Copies
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing 13.1)

Uncovering a problem

Now the test is passing, and happy customers confirm that they're not able to buy non-existent books, so let's take this opportunity to review our test coverage:

```
go test -cover
PASS
coverage: 75.0% of statements
```

Oh dear, that's a little lower than we'd like. Fully 25% of our code is untested. Let's use the coverage tools to identify which lines are the problem (either by generating HTML or by using our editor's built-in coverage highlighting). Here they are:

```
    return Book{}, errors.New("no copies left")
}
```

So our `if` statement is executed by the test; that makes sense, because it will always compare `b.Copies` with zero, whatever's passed in. But the `return` statement that returns an error value is never executed by any test.

That doesn't mean it's incorrect, only that it's not executed by any test. But that's a risky situation to be in, because even if it happens that the code is correct *now* (and it is), there's no guarantee that someone won't come along and modify it later, breaking its behaviour. They'll run the tests to satisfy themselves that nothing is wrong, and *they will pass*.

We should fix that right away by adding a test for the new behaviour. What's the behaviour? Well, if you try to buy a book with no copies, you should get an error. So let's write that:

```
func TestBuyErrorsIfNoCopiesLeft(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title:  "Spark Joy",
        Author: "Marie Kondo",
        Copies: 0,
    }
    _, err := bookstore.Buy(b)
    if err == nil {
        t.Error("want error buying from zero copies, got nil")
    }
}
```

(Listing 13.1)

This passes, and we're back to 100% coverage, so we can breathe a sigh of relief.

"Covered" versus "tested"

Untested code is a problem waiting to happen, so you can see why many people routinely check their test coverage numbers, and even sometimes require a minimum percentage value before code can ship.

But there are a couple of caveats to be aware of. One is that just because a line of code is *executed* by a test, that doesn't mean that the test necessarily tests anything useful about that code's behaviour.

For example, we could trivially *cover* the `Buy` function with a test like this:

```
func TestBuyTrivial(t *testing.T) {
    t.Parallel()
    bookstore.Buy(bookstore.Book{})
```

```
    bookstore.Buy(bookstore.Book{Copies:1})
}
```

This test brings our coverage numbers up to 100%, but it's useless all the same: it doesn't verify anything about the behaviour of `Buy`. "Covered" isn't the same as "tested". If you remember the principle of "test behaviours, not functions", you won't fall into this trap.

Juking the stats

The other thing to beware of is "chasing coverage": trying to make the numbers bigger just for the sake of the numbers. This is understandable in an environment that requires or rewards a high percentage of test coverage, but as we just saw, even 100% coverage doesn't guarantee correct behaviour. (It does at least tell you that the code compiles, but perhaps there are easier ways of checking that.)

It's rare in real Go programs for our coverage to be 100%. But why? Consider a piece of code like this:

```
f, err := os.Open("some file")
if err != nil {
    return err
}
```

Any test that calls this code will probably not show the `return err` line as covered, so overall our coverage will be less than 100%. Is that a problem? Not necessarily.

We *could* test it: perhaps by writing a test that tries to pass in a non-existent file. That would improve the coverage numbers, but what are we really *testing* here? Nothing but that `if` statement.

Again, the "behaviours, not functions" principle is helpful. If there's no real *behaviour* in the untested code, it's probably not worth testing. We can see it's correct just by looking.

So we don't need 100% test coverage, and even when we have it, it doesn't guarantee that the code is actually correct. 80-90% coverage is probably a reasonable figure to aim for.

Takeaways

- A good way to start thinking seriously about what a package needs to do is to write some *user stories*: brief descriptions of some interaction with the program from the user's point of view.
- User stories are not the same thing as *features*, which are just "things the program

can do": a given user story might make use of several features, while some features might not affect users at all.

- For any given program, it's helpful to start by identifying the *core stories*, the smallest set of user stories that render the program useful at all, and we should aim to implement these stories first.
- When designing the code to implement a user story, it helps to think in terms of *behaviours* rather than *functions*: first ask, "How does the program need to behave?", and when you know the answer to that, you'll have a better idea of what functions you might need to write.
- It's worth taking a little trouble over your test failure messages: include as much information as you can about what the test did, what was supposed to happen, what *actually* happened, and what kind of problem that indicates.
- We can set and get the values of struct fields by using the dot notation: for example, `b.Copies` refers to the `Copies` field of the struct `b`.
- When we're adding or subtracting numbers from a variable, we can use the `+=` and `-=` operators: `b.Copies -= 5`.
- And when we just want to add or subtract exactly 1 (which is quite common), we can use the `++` and `--` operators: `b.Copies--`.
- A given test *covers* some particular set of code lines: that is, when we run the test, those lines are executed.
- The Go tools can tell us what *percentage* of our code is covered by tests, and show what is and isn't covered, which can be helpful for spotting things we forgot to test.
- If we're testing each behaviour before we implement it, then there simply can't *be* any important code that isn't covered by tests, but if anything does slip through this net, the code coverage tool will help us find it.
- Any time you have an `if` statement in code, there are two possible code paths, representing two different behaviours, and we may well need a test for both of them.
- We should be careful not to read too much into test coverage statistics: they only prove that a given line is *executed* by a test, not that it actually *works*.
- Generally speaking, more test coverage is better than less, but don't aim for some specific number, because that's no use: instead, aim to test 100% of important *behaviours*, rather than 100% of *lines*.
- It's okay for the coverage to be less than 100% when the uncovered lines are trivial, when we can check their correctness by eye, and when there's no real value in writing a test for them: this often applies to simple `if err != nil` blocks.

6. Slicing & dicing

You've made a great start on our bookstore project so far, but there's something we're still missing. We can create a single value of the `Book` type and assign it to a variable, but a decent bookstore is going to need more than one book. We don't yet have a way of dealing with *collections* of things (books, for example).

Slices

Just as we grouped a bunch of values of different types together into a struct, so that we can deal with them as a unit, we would like a way to group together a bunch of values of the *same* type. And you'll recall from earlier in the book that in Go one kind of collection of values is called a *slice*.

For example, a value representing a group of books would be a *slice of `Book`* (that's how we say it). We write it using empty square brackets before the element type: `[]Book`. Just as each individual `struct` type in Go is a distinct type, so is each slice.

Slice variables

Can we declare variables of a slice type? Yes, we can:

```
var books []Book
```

This says "Please create a variable named `books` of type `[]Book`" (slice of `Book`).

Slice literals

We've seen how to write struct literals; you might remember from your `calculator` tests that we can also write a *slice literal*:

```
books = []Book{}
```

The value on the right-hand side of the assignment is an empty `[]Book` literal. If we wanted to include some actual books in it, we could do that, by putting some `Book` literals (each with a trailing comma) inside the curly braces:

```
books = []Book{
    {Title: "Delightfully Uneventful Trip on the Orient Express"},
    {Title: "One Hundred Years of Good Company"},
}
```

Notice that we don't need to write the type name `Book` before each struct literal inside the slice, because Go knows that only `Book` values can possibly be members of a slice of `Book`.

Slice indexes

There's something we can do with a slice that we couldn't do with a basic or struct type: we can refer to one of its *elements*. If we just want, for example, the first book in the slice, we can specify which element we want using its *index* in square brackets:

```
first := books[0]
```

The index is a non-negative integer that specifies the single element we're interested in, where the index of the first element is 0, the second is 1, and so forth. (Think of the index as an *offset* from the start of the slice, if that helps.)

Slice length

Given a slice variable of a certain length, then, how can we tell which indexes are safe to use? That's equivalent to knowing how many elements are in the slice, and there's a built-in function `len` (short for "length") for that:

```
fmt.Println(len(books))
// 2
```

This is telling us that the `books` slice has 2 elements.

Modifying slice elements

Can we modify an individual slice element directly, using its index? It turns out we can:

```
books[0] = Book{Title: "Heart of Kindness"}
```

Indeed, we can use the same trick as with structs, to modify an individual *field* of an individual element:

```
books[0].Title = "Heart of Kindness"
```

Appending to slices

There's something else we can do with a slice that we couldn't do with our other types: we can *add a new element* to it. We use the built-in `append` function for this:

```
b := Book{ Title: "The Grapes of Mild Irritation" }
books = append(books, b)
```

`append` takes a slice and an element value (or multiple values), and returns the modified slice, with the new value as its last element.

A collection of books

Let's implement another user story for the bookstore: listing all books. Suppose we implement this with a function called `GetAllBooks`. How shall we write a test for it? What's the behaviour we want?

Well, we need some concept of "the collection of books in stock". We can now make a guess that a *slice* of books will be a useful way to represent this collection. How do we spell that in Go? Its type will be `[]Book`.

So there needs to be some `[]Book` variable that we will use to store the books we have in stock: our *catalog*. Let's call this variable `catalog` (it's a good idea to always use the same name for the same variable, wherever it occurs in your program).

We're starting to put together the pieces we need: a slice variable `catalog`, and a function `GetAllBooks`. Whatever is contained in `catalog`, that's what `GetAllBooks` should return. Let's start sketching out a test along those lines.

Setting up the world

We know we'll need to call `GetAllBooks` in the test, pass in the `catalog` variable, representing our catalog of books, and check its return value against some expectation. That suggests we need to add some books to the catalog first.

Well, we've said `catalog` is a slice variable, so we might start by assigning it a slice literal containing some books:

```
catalog := []bookstore.Book{
    {Title: "For the Love of Go"},
    {Title: "The Power of Go: Tools"},
}
```

(Listing 15.1)

What's our `want`? Well, exactly the same slice literal:

```
want := []bookstore.Book{
    {Title: "For the Love of Go"},
    {Title: "The Power of Go: Tools"},
}
```

(Listing 15.1)

As usual, we'll call the function under test to get our `got`:

```
got := bookstore.GetAllBooks(catalog)
```

(Listing 15.1)

Finally, we're ready to compare `want` with `got` to see if the function returned the correct result.

Comparing slices (and other things)

There's just one problem. Usually in a test we compare `want` with `got`, to see if it was as expected. But we can't actually compare two slices in Go using the == operator:

```
if want != got {
// invalid operation: want != got (slice can only be compared
// to nil)
```

The standard library function `reflect.DeepEqual` can do this comparison for us, but there's an even better solution. There's a package called `go-cmp` that is really clever at comparing all kinds of Go data structures, and it's especially useful for tests.

Add this to your imports in the test file:

```
"github.com/google/go-cmp/cmp"
```

To download the package, you'll need to run:

```
go get -t
```

This tells the go tool to download all packages required to build your tests:

```
go get: added github.com/google/go-cmp v0.5.6
```

You can now use the `cmp.Equal` function, which returns `true` if its two arguments are equal, or `false` otherwise:

```
if !cmp.Equal(want, got) {
```

What's even handier is that if the two things *aren't* equal in some way, we don't have to just print them both out in full and play "spot the difference". Instead, `cmp.Diff` will show the differences element by element, line by line, just like the Unix `diff` command.

Let's try it with two slices of `string`:

```
want := []string{"same", "same", "same"}
got := []string{"same", "different", "same"}
if !cmp.Equal(want, got) {
    t.Error(cmp.Diff(want, got))
}
```

The output looks like this:

```
   []string{
    "same",
-   "same",
+   "different",
    "same",
   }
```

See how it's highlighted the one element that differs between the two slices? The expected value, from `want`, is marked with a `-` prefix, and the value actually received is shown prefixed by `+`.

In other words:

```
want "same"
got "different"
```

So if you're comparing two values of a basic type in a test, you can use `==`, but for structs, slices, or anything more complicated, you'll find `cmp.Diff` and `cmp.Equal` indispensable.

Here's the complete test, ready to run:

```
func TestGetAllBooks(t *testing.T) {
    t.Parallel()
    catalog := []bookstore.Book{
        {Title: "For the Love of Go"},
        {Title: "The Power of Go: Tools"},
    }
```

```
    want := []bookstore.Book{
        {Title: "For the Love of Go"},
        {Title: "The Power of Go: Tools"},
    }
    got := bookstore.GetAllBooks(catalog)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing 15.1)

It's not compiling yet, nor even failing—but I'm sure you can take it from here.

GOAL: Add the minimum code necessary to get this test to compile and fail. Then make it pass.

As long as your solution passes the test, it's fine. Just for fun, let's see what the failure looks like with a null implementation of `GetAllBooks` (that is, one that returns an empty slice, or `nil`):

```
[]bookstore.Book(
    - {{Title: "For the Love of Go"},
        {Title: "The Power of Go: Tools"}},
    + nil,
)
```

You now know how to interpret this `cmp.Diff` output correctly. What it expected (the `want` value) is the slice literal shown with a leading - sign:

```
- {{Title: "For the Love of Go"},
    {Title: "The Power of Go: Tools"}},
```

But what it *got* (shown with a leading + sign) was `nil`, hence the failure.

We should be able to make this pass by returning the contents of the `catalog` parameter:

```
func GetAllBooks(catalog []Book) []Book {
    return catalog
}
```

(Listing 15.1)

Indeed, this passes the test. Well done!

Unique identifiers

Let's move on to the third user story on our list: getting details of a particular book. So what's the behaviour we want to test?

Suppose there were some function `GetBook` that, given a catalog of books, returns the details of a specific book. What result type would make sense? Well, we already have a data type that stores all information about a book: the struct `Book`. So that part is easy: `GetBook` will return a `Book` value.

So what parameter, apart from the catalog to search, should `GetBook` take? Well, that's a slightly more tricky question. How do we uniquely identify a book? Not by its title, because there can be many books with the same title. Not by its author, price, or any of the other fields on `Book`, either, because none of them are unique. There is such a thing as an International Standard Book Number (ISBN), but not every book has one; the one you're reading doesn't, for example!

So we're going to need some unique identifier (ID) for each book. To be clear, it doesn't matter what that value actually *is* for any given book, only that each value uniquely identifies a distinct book.

What type would make sense for the `ID` field, then? Perhaps `int`; we could think of this as the book's catalog number, for example.

So let's say that if we add a field `ID int` to our `Book` type, we can use that as the unique ID for each book. Accordingly, `GetBook` should take a book ID as its second parameter (that is, an `int` value representing the ID of the book you're interested in).

To make things easy, we'll assume here that each book already has an ID assigned to it, but for a fun thought experiment, you might like to think about how to generate book IDs automatically, in a way that ensures they're always unique.

Getting to a failing test

We can now start to write `TestGetBook`. Suppose we added some book with ID 1 to the catalog. We could call `bookstore.GetBook(catalog, 1)` and check that the book returned is the same one we started with.

GOAL: Write `TestGetBook` along these lines.

Once you're done, you should be getting compile errors because the `Book` struct doesn't have a field `ID`, and because `GetBook` doesn't exist yet.

GOAL: Write the minimum code necessary to get `TestGetBook` to compile and fail.

Because `GetBook` doesn't yet return the right answer, you should see a test failure similar to this:

```
bookstore_test.go:18:   bookstore.Book{
    -       Title:  "For the Love of Go",
```

```
    +       Title:  "",
            Author: "",
            Copies: 0,
    -       ID:     1,
    +       ID:     0,
      }
```

This is saying it wanted the title of the returned book to be `"For the Love of Go"`, but it was in fact the empty string, `""`. Similarly, it wanted the ID of the book to be `1`, but it was in fact `0`, because an empty `Book` struct has the default values of all its fields: empty string for a `string` field, and `0` for an `int` field.

Here's my version of the test, but anything that produces a similar failure will be fine:

```
func TestGetBook(t *testing.T) {
    t.Parallel()
    catalog := []bookstore.Book{
        {
            ID: 1,
            Title: "For the Love of Go",
        },
    }
    want := bookstore.Book{
        ID: 1,
        Title: "For the Love of Go",
    }
    got := bookstore.GetBook(catalog, 1)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing 15.2)

In order for this test to compile, we need to add an `ID` field to our `Book` type:

```
type Book struct {
    Title  string
    Author string
    Copies int
    ID     int
}
```

(Listing 15.2)

Finding a book by ID

Nicely done. You're ready to go ahead and write the real implementation of `GetBook`. But how are we going to find the book in the catalog that has the ID we want?

GOAL: Think about this a little and see if you can work out a way to do it, before reading on. It doesn't have to be a *good* way; our bookstore is still at the prototype stage, for now. We just need to get the test passing.

Well, the simplest way that could possibly work is to look at every book in the catalog to see if it's the right one! That's not particularly elegant, or even scalable, but it'll do for now.

Remember the `for ... range` statement from the `calculator` project, earlier in this book? We used it to loop over a slice of test cases. Since we need to look at each book in the catalog in turn, a similar loop might work here.

Here's what that might look like:

```
for _, b := range catalog {
```

Here, we're ranging over a slice of books instead of a slice of test cases, but it's the same kind of idea. We'll do something once for each element of the slice `books`, and each time round the loop, the variable `b` will be each successive book in the slice.

Loops in Go using the `for` keyword are a very powerful control structure, and we'll look at them in much more detail later on, in the "Switch which?" chapter. For now, though, let's just *use* this loop and see what we can do with it.

GOAL: Implement `GetBook` using a `range` loop.

Let's start with the loop statement we already have:

```
for _, b := range catalog {
```

What should we do for each `b`, then? Well, we're interested only in the `b` whose ID is the one we want:

```
if b.ID == ID {
```

If this is true, then we've found the book we're looking for. What should we do? Well, return it:

```
return b
```

What happens if we reach the end of the loop without finding the book? Well, we have to return *something*, and at the moment the only thing that makes sense is an empty `Book` value:

```
return Book{}
```

So here's a complete, working version of `GetBook`:

```
func GetBook(catalog []Book, ID int) Book {
    for _, b := range catalog {
        if b.ID == ID {
            return b
        }
    }
    return Book{}
}
```

(Listing 15.2)

This passes the test.

Crime doesn't pay

This test has at least forced us to do *something* about finding books by ID, but is it sufficient to give us real confidence that `GetBook` is correct?

One way to answer this question is to ask "Is there an incorrect implementation of `GetBook` that would still pass this test?"

Well, here's one:

```
func GetBook(catalog []Book, ID int) Book {
    return catalog[0]
}
```

Oh dear. This clearly isn't right: it always just returns the *first* book in the catalog, whether that's the one you wanted or not. But it passes the test, because in the test, there *is* only one book in the catalog.

How can we improve the test so that we can't pass it with such a silly version of `GetBook`? Yes, you guessed it: add *another* book to the test catalog. Then call `GetBook` with the ID of the *second* one. This still isn't completely error-proof, but it's better.

GOAL: Update your test so that it adds two books to the catalog, and checks that it can retrieve the second one (ID 2).

This gives us a lot more confidence in `GetBook`. We can never be absolutely certain that the implementation is correct, but it's helpful to think about possible ways it could be wrong and improve our tests to catch them.

Here's my updated test:

```
func TestGetBook(t *testing.T) {
    t.Parallel()
    catalog := []bookstore.Book{
```

```
        {
            ID: 1,
            Title: "For the Love of Go",
        },
        {
            ID: 2,
            Title: "The Power of Go: Tools",
        },
    }
    want := bookstore.Book{
        ID: 2,
        Title: "The Power of Go: Tools"
    }
    got := bookstore.GetBook(catalog, 2)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing 15.3)

Takeaways

- A *slice* is a kind of Go type that represents an ordered collection of *elements*, all of the same type: for example, `[]string` is a slice of strings.
- A *slice literal* specifies the name of the slice type, followed by some elements in curly braces: for example, `[]string{"a", "b", "c"}`.
- In a slice literal of some custom type, such as a struct, we don't need to repeat that type name as part of every element literal: the compiler can infer it.
- We use the square-bracket notation to refer to a particular numbered element of the slice: for example, `books[0]` refers to the first element of the `books` slice.
- The built-in function `len` tells us the number of elements in a slice.
- The built-in function `append` appends one or more elements to a slice, and returns the modified slice.
- The `range` operator lets us write `for` loops that execute once for each element of the slice in turn.
- In a loop like `for _, b := range books`, each time round the loop `b` will be each successive element of the slice `books`.

- The == operator isn't defined on slices, so we can use the `go-cmp` package instead, which can compare all sorts of values using `cmp.Equal`, and produce a *diff* with `cmp.Diff` if they're not equal.
- The output from `cmp.Diff` shows what was *expected* with a leading - sign, and what was actually *received* with a leading + sign.
- When you need to retrieve some specific thing from a collection of things, you need a way of uniquely identifying it by an *ID*, which can be whatever you want, but is often a number.
- One way to get a specific element from a slice, if you don't already know its index, is to loop over the whole slice comparing each element with the one you want.

7. Map mischief

Thanks to you, Happy Fun Books is already taking shape! We can store a catalog of books stocked by each store, get the list of all books in the catalog, and look up details on a specific book by ID.

A successful bookstore, though, will have lots of books, and it could take a long time to loop through the whole list comparing IDs one by one. Can't we do better?

What we want is a data structure that, given a book ID, returns the corresponding book directly, without needing to loop. This would be a direct *mapping* from IDs to books.

Introducing the map

It turns out that Go has exactly the data type we want. And, for reasons that may now be clear to you, it's called a *map*.

Remember how a slice is a collection of values, each of which is uniquely identified by a number? A map is similar, but instead of a number, we can use a *key* that can be a value of any Go type. Here, our keys will be the book IDs, which are integers, so our map key type will be `int`.

We should be able to rewrite `TestGetBook` in such a way that it sets up the `catalog` variable to be a map instead of a slice, shouldn't we? Let's try.

The first thing we do in the test as it stands now is to assign a slice literal to `catalog`:

```
catalog := []bookstore.Book{
    {ID: 1, Title: "For the Love of Go"},
    {ID: 2, Title: "The Power of Go: Tools"},
}
```

(Listing 15.3)

How would that change if we're assigning a map literal instead? Not that much, it turns out:

```
catalog := map[int]bookstore.Book{
    1: {ID: 1, Title: "For the Love of Go"},
    2: {ID: 2, Title: "The Power of Go: Tools"},
}
```

(Listing 16.1)

The type of this literal is `map[int]bookstore.Book` (pronounced "map of int to bookstore dot book"). As you might expect, this sets up a mapping between some integer (the book ID) and some `Book` value, so that we can retrieve the `Book` directly by its ID.

You can see that the individual `Book` literals are the same as before. What's changed is that now there's an ID, followed by a colon, before each `Book` value:

```
1: {ID: 1, Title: "For the Love of Go"},
```

(Listing 16.1)

Here, the key is `1`, and the value is the `Book` literal with that ID. Just as with a slice literal, we need a trailing comma after every element in a map literal.

And the rest of the test doesn't need to change at all. This is a good sign: tests shouldn't be too tightly coupled to implementation details. If they are, that means your public API is coupled to those details too, so that you can't change one without affecting the other. Instead, a useful abstraction *conceals* details that users shouldn't need to care about.

What do we need to change in the `bookstore` package to make this work? Let's take a look at the current version of `GetBook`:

```
func GetBook(catalog []Book, ID int) Book {
    for _, b := range catalog {
        if b.ID == ID {
            return b
        }
    }
    return Book{}
}
```

(Listing 15.2)

We know we won't need this `range` loop anymore: a map lets us look up the book we want directly, without looping. And to do that, we can use a syntax very similar to a slice index expression:

```
func GetBook(catalog map[int]Book, ID int) Book {
    return catalog[ID]
}
```

(Listing 16.1)

Notice the square brackets? The key goes between those brackets in just the same way as the index of a slice, and the result of this expression is the element value.

Adding a new element to a map

Recall from the chapter on slices that we can update a slice by assigning a new element to a specific index. Here's how we assign a new value to a map:

```
catalog[3] = Book{ID: 3, Title: "Spark Joy"}
```

If you do this with a key that already exists in the map, the new element will overwrite the old one. So each key is unique: you can't have duplicate keys in the same map. (You can have duplicate *elements*, though, provided each is identified by a different key.)

Accessing element fields

If you just want to read the value of an element's field (for example, a book's `Title`), you don't need to assign the whole element to a variable first. You can refer to the field directly like this:

```
fmt.Println(catalog[1].Title)
// For the Love of Go
```

This expression is in two parts. First, a map *index expression*, identifying the element you're interested in (ID number `1`). Next comes the field *selector*, naming the specific field of the element that you want, preceded by a dot: `.Title`.

So given the expression `catalog[1].Title`, Go first of all looks up the `1` key in the `catalog` map, gets the `Book` element it identifies, and gives you the value of its `Title` field, which is `For the Love of Go`.

Updating elements

What if we don't want either to add a new book to the map, or to completely overwrite an existing one? Instead, we might like to modify just one field of a given book. How can we do that?

Since you know that you can *read* the value of a struct field in a map directly, as we saw in the previous section, you might wonder if you could also *set* it in the same way:

```
catalog[1].Title = "For the Love of Go"
// this doesn't work
```

The compiler will complain about this:

```
cannot assign to struct field catalog[1].Title in map
```

Unfortunately, Go doesn't allow you to modify fields of map elements directly like this. Instead, we have to get the element out of the map, assign it to some variable, and modify that variable instead. We can then use the modified value to overwrite the original book:

```
b := catalog[1]
b.Title = "For the Love of Go"
catalog[1] = b
```

Non-existent keys

It might have already occurred to you, looking at the test for `GetBook`, to wonder what happens if you try to look up a key that isn't, in fact, in the map?

Suppose we accidentally on purpose got the ID wrong in the test, and tried to look up ID 3 instead of ID 2:

```
got := bookstore.GetBook(catalog, 3)
```

We might expect this to trigger an `index out of range` panic, as it would when looking up a slice index that doesn't exist. But that's not the case. Instead, we just get a test failure:

```
bookstore_test.go:19:   bookstore.Book{
    -       Title:  "The Power of Go: Tools",
    +       Title:  "",
            Author: "",
            Copies: 0,
    -       ID:     2,
    +       ID:     0,
        }
```

It seems that `GetBook(catalog, 3)` returns a perfectly valid `Book` value, but it corresponds to a mysterious book with no title and with ID 0. What's going on?

An interesting property of Go maps is that looking up a non-existent key doesn't cause an error: instead, it returns the zero value of the element type. Recall that the zero value of a struct is an instance of that struct whose fields all have *their* zero value.

This is the same kind of failure that we saw with our null implementation of `GetBook` that returned just `Book{}`, isn't it? And since `GetBook` returns whatever the value of `catalog[ID]` is, in this case it will be just `Book{}`. So if you call `GetBook` with a non-existent book ID, you get an empty `Book` struct.

We can trivially fix the test by undoing the deliberate mistake of asking for ID 3 instead of 2, but this experiment suggests an interesting thought. How can we tell whether or not a given book ID is in the map? That is to say, in the catalog. What should happen, for example, if a user tries to search for a book ID that doesn't exist? Should we show them a page of book details where every field is empty?

That doesn't sound very friendly. It would be nice if `GetBook` had some way of indicating, along with the returned `Book` value, whether or not the requested book actually exists.

We could look at the `Book` and check if all its fields are zero, or checking if the value is equal to `Book{}`, but that seems a little hacky. Is there a better way?

Checking if a value is in the map

There's a neat extension to the normal map lookup syntax that can give us exactly the information we need. Usually, we just receive one value from a map lookup, like this:

```
b := catalog[ID]
```

But we can receive a second value too, conventionally named `ok`:

```
b, ok := catalog[ID]
```

The second value is of type `bool`, and it's true if the given key was found in the map, but false otherwise.

Let's see how to apply this idea to our `GetBook` function. We now have a powerful data structure to represent our book catalog: a map of IDs to books. We've also asked the question: what should happen when you try to look up details on a book ID that doesn't exist, and how can we determine when this is the case?

Returning an error

Earlier we saw that the second value received from a map lookup, conventionally named `ok`, tells us whether or not the key was in the map.

How can we use this to improve the usability of our `GetBook` function? Well, one idea would be to have `GetBook` just return the `ok` value along with the book. Then anyone who wants to know if the ID existed or not can check that returned value.

Let's think more generally, though. You already know from the `calculator` package that Go has a standard way of indicating that a function's input is invalid, or that some other problem occurred: returning an `error` value. In the case where the book doesn't exist, we can return a helpful error message pointing that out.

How would this change our test for `GetBook`? Well, the first thing we'd need to do is receive two values from `GetBook` instead of one:

```
got, err := bookstore.GetBook(catalog, 2)
```

(Listing 17.1)

As before, if `err` is non-nil then `got` is invalid (and we should fail the test). And, just as with `TestDivide`, an error at this point in the test is *fatal*: we should stop the test altogether and not bother checking anything else:

```
if err != nil {
    t.Fatal(err)
}
```

(Listing 17.1)

This won't compile yet, because we need to make some tweaks to the `GetBook` code to reflect the new function signature:

```
func GetBook(catalog map[int]Book, ID int) (Book, error) {
    return catalog[ID], nil
}
```

The test now passes! Are we done? Not quite, because we've still only tested the valid input case.

GOAL: Test the invalid input case.

Testing the invalid input case

Just as you did in the `calculator` project, start by defining a new test, called something like:

```
func TestGetBookBadIDReturnsError(t *testing.T) {
    t.Parallel()
    catalog := map[int]bookstore.Book{}
    _, err := bookstore.GetBook(catalog, 999)
    if err == nil {
```

```
        t.Fatal("want error for non-existent ID, got nil")
    }
}
```

(Listing 17.1)

And that's it. Unsurprisingly, since `GetBook` always returns a nil error, this test won't pass.

GOAL: Make this test pass.

This is a little trickier, but you have all the pieces you need to put this together. If you're having trouble, review the information in this chapter, and keep trying. When *both* tests pass, you're done!

Let's work our way through the solution. A single `return` statement in `GetBook` won't be good enough anymore, since we have two possible code paths.

That suggests we'll need an `if` statement to choose between them. First of all, let's get the values we need to make that decision:

```
b, ok := catalog[ID]
```

The `ok` value tells us whether or not the requested ID was actually in the map. If it's false, we can return an empty `Book` along with a suitable error message. Otherwise, we return the `b` that we successfully retrieved from the catalog:

```
func GetBook(catalog map[int]Book, ID int) (Book, error) {
    b, ok := catalog[ID]
    if !ok {
        return Book{}, fmt.Errorf("ID %d doesn't exist", ID)
    }
    return b, nil
}
```

(Listing 17.1)

Notice one little refinement we made in the error message here. It's always helpful in "invalid input" cases to say what the invalid data actually *was*. Since we have the ID value, we can include it in the message, using `fmt.Errorf` instead of `errors.New`.

Updating GetAllBooks

We've been moving so fast we haven't yet had a chance to write an updated version of `GetAllBooks` that works with the new map type. Let's take care of that now. First of all, let's update the test to create the catalog as a map:

```
func TestGetAllBooks(t *testing.T) {
    t.Parallel()
    catalog := map[int]bookstore.Book{
        1: {ID: 1, Title: "For the Love of Go"},
        2: {ID: 2, Title: "The Power of Go: Tools"},
    }
    want := []bookstore.Book{
        {ID: 1, Title: "For the Love of Go"},
        {ID: 2, Title: "The Power of Go: Tools"},
    }
    got := bookstore.GetAllBooks(catalog)
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing 15.1)

We don't expect this to pass yet. `GetAllBooks` still tries to simply return the `catalog` parameter it's passed, and that won't work now:

```
bookstore.go:14:2: cannot use catalog (type map[int]Book) as
type []Book in return argument
```

We'll need to do a little more work in `GetAllBooks` now to construct the required slice.

GOAL: Make the test pass. If you're having trouble, review the chapter so far; this may give you some useful ideas.

Our first guess at this might be to start with an empty slice, and loop over the map, appending each element to the slice as we go. At the end of the loop, the slice should contain all the elements of the map. Here's what that looks like:

```
func GetAllBooks(catalog map[int]Book) []Book {
    result := []Book{}
    for _, b := range catalog {
        result = append(result, b)
    }
    return result
}
```

(Listing 17.2)

We start by creating the variable `result` as an empty slice of `Book`. Then we use `range` to loop over the `catalog` map. Recall that `range` over a map returns two values each

time, the key and the element. In this case the key is the book ID, and we don't need that, so we ignore it using the blank identifier `_`.

Each time through the loop, we append the next element value to our result, and finally we return the result. This should be a slice containing every book in the map, which is what the test expects. Does the test pass?

Yes, it does. Encouraging! Let's run the test again just to enjoy our moment of triumph a little longer.

```
--- FAIL: TestGetAllBooks (0.00s)
    bookstore_test.go:22:   []bookstore.Book{
            {
    -               Title:  "For the Love of Go",
    +               Title:  "The Power of Go: Tools",
                    Author: "",
                    Copies: 0,
    -               ID:     1,
    +               ID:     2,
            },
            {
    -               Title:  "The Power of Go: Tools",
    +               Title:  "For the Love of Go",
                    Author: "",
                    Copies: 0,
    -               ID:     2,
    +               ID:     1,
            },
        }
```

Wait, what? Didn't that test pass just now? Let's run it again.

```
PASS
```

Something weird is going on. Digging into this a little more, it seems that roughly half the time, `GetAllBooks` returns the exact slice we expect, thus passing the test. But the rest of the time, the slice elements are in a back-to-front order, with book ID 2 listed first. Since slices are ordered, this doesn't compare equal with our `want` slice, and so the test fails.

In fact, we can reproduce this strange behaviour with a much simpler loop:

```
for ID := range catalog {
    fmt.Println(ID)
}
```

The output from this is:

```
1
2
```

Or, about half the time:

```
2
1
```

Clearly the map isn't changing from one run of this program to the next, so it appears that the `range` loop is enumerating the keys of the map in a changing order. Is there something wrong with Go itself?

In fact, this behaviour is by design. Slices in Go are *ordered*, meaning that they're not just a bunch of elements. They come in a specific *sequence*. But maps aren't like that: they are, in some sense, just a bunch of key-value pairs, without any particular order to them.

Our test uses a map of two books, giving two possible orderings, hence the test fails about half the time. What can we do to avoid this?

Well, let's think a bit more carefully about *what we're really testing here*. When we're getting the list of all books, do we care that they're in a particular order? Not really. So `GetAllBooks` is actually doing the right thing: returning the complete list of books, in no particular order. The problem, then, is with our *test*.

We're comparing the result of `GetAllBooks` against a slice, which, as we've seen, is inherently ordered. But we don't care what order we see the books in, so long as they're all present. To make the test reliable, then, we could *sort* the result slice before comparing it against our expectation.

Sorting slices

The standard library provides a very handy `sort.Slice` function that will do just what we need. It takes two arguments: the slice to be sorted, and a comparison function that explains how any two elements of the slice should be ordered.

The comparison function tells `sort.Slice` *how* to sort the elements. That is, given two elements (let's call them `i` and `j`), the comparison function determines if element `i` should come before element `j`. It returns `true` if this is the case.

How should we sort our slice of books? Well, it doesn't really matter, so long as they end up in the same order as our `want` slice, and that happens to be in ID order. So let's do the same here:

```
sort.Slice(got, func(i, j int) bool {
    return got[i].ID < got[j].ID
})
```

(Listing 17.3)

Function literals

There's some new syntax here: a *function literal*. You've seen literals of various kinds before: strings, structs, slices, and maps. The second argument to `sort.Slice` is a function literal. It's written like this:

```
func(i, j int) bool {
    return got[i].ID < got[j].ID
}
```

As with the kind of named functions you've defined before, it starts with the `func` keyword, to say "here comes a function". But we omit the name that would usually come next, because this is an *anonymous* function. We only need it this once, and it won't be referred to again, so giving it a name would be a waste of time.

Instead, the usual list of parameters and results comes next. In this case, the function takes two integer parameters `i` and `j`, and returns a `bool` result. The function body itself follows, inside curly braces.

The comparison function is easy to write in this case: we just need to supply a boolean expression that determines whether element `i` should be sorted before element `j`. Since we've decided to sort by ID, our expression compares the ID field of the two `Book` values `i` and `j`:

```
got[i].ID < got[j].ID
```

The effect of this call to `sort.Slice` is to sort the `got` slice, in place, by book ID. Since our `want` slice is already sorted by ID, the two slices should always compare equal, and the test should thus pass reliably. Let's try a few times to make sure:

```
go test
```

```
PASS
ok      bookstore       0.192s
```

```
go test
```

```
PASS
ok      bookstore       0.148s
```

```
go test
```

```
PASS
ok      bookstore       0.149s
```

Great! No more mysterious intermittent failures.

The final test, including the slice sorting, looks like this:

```
func TestGetAllBooks(t *testing.T) {
    t.Parallel()
```

```
    catalog := map[int]bookstore.Book{
        1: {ID: 1, Title: "For the Love of Go"},
        2: {ID: 2, Title: "The Power of Go: Tools"},
    }
    want := []bookstore.Book{
        {ID: 1, Title: "For the Love of Go"},
        {ID: 2, Title: "The Power of Go: Tools"},
    }
    got := bookstore.GetAllBooks(catalog)
    sort.Slice(got, func(i, j int) bool {
        return got[i].ID < got[j].ID
    })
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing 17.3)

Takeaways

- When we want to retrieve a specified element from a collection *directly*, without looping to look for it, a *map* is a good choice of data structure.
- A map links *keys*, such as book IDs, to elements, and a map literal comprises a list of key-element pairs.
- Looking up a map element by key uses the same square-bracket notation as looking up a slice element by index: for example `catalog[ID]`.
- We can also assign a new element to a specified map key; if the key already exists, its old element will be replaced with the new one, and if it doesn't exist, the new element will be added to the map.
- While you can *refer* to some field of a struct element in a map using the usual dot notation, you can't *assign* to it: instead, you have to look up the element, store it in a variable, modify its field, and then write it back to the map.
- The result of looking up a key that doesn't exist in the map is the zero value for the element type: for example, with a map of `Book`, if you look up some key that's not in the map, you'll get an empty `Book` value.
- If we receive a second `ok` value from the map lookup, it will tell us whether the key actually existed (`true`), or whether we're just getting a default zero value instead (`false`).

- For functions like `GetBook` that retrieve a specified value, it makes sense to return `error` if the value doesn't exist.
- And to make the error message more helpful, we can use `fmt.Errorf` to interpolate some data into it, such as the ID the user asked for.
- If we want a slice containing all the keys of some map, we can loop over the slice with `for ... range`, appending each key to our result slice.
- But the keys will be in a different, random order each time the program runs; this is by design, because maps just don't *have* any defined key order.
- So if you don't *care* about the order, which you shouldn't if you're using a map, then don't *rely* on it, because it will keep changing.
- To simplify testing functions that return items from a map, we can use `sort.Slice` to sort the results so that we can compare them reliably.

8. Objects behaving badly

In the previous chapters, we've learned a bunch of useful things about working with *data* in Go (for example, data about books) and how to represent that data using types, variables, and so on.

Let's continue that journey now as we learn more about what we can do with data: specifically, giving it *behaviour*. That might sound a bit weird (how can data have behaviour?), but stay with me. First, let's talk about *objects*.

Objects

It's quite common to deal with things in our programs that represent objects or entities in the real world. For example, in the bookstore application, we have a `Book` type that represents a book. We could also imagine objects like a `Customer`, an `Order`, and so on. We use Go's `struct` keyword to define types to represent objects like this in Go programs. Here's an example you've already seen:

```
type Book struct {
    Title  string
    Author string
```

```
    Copies int
    ID     int
}
```

(Listing 15.2)

These objects have *properties* (that is, facts about them), and in Go we represent these as *fields* on the struct type. For example, a book has an author, and correspondingly, the `Book` struct has an `Author` field where we can store this information.

In theory, we could model any real-world problem as a collection of objects that have various properties, and relate to each other in various ways. For example, a drawing program might deal with Shape objects, that might store information about their vertices or edges, or perhaps Brush objects with different properties. A card game program might have objects like Card, Hand, and Player.

Or maybe our program could use objects that don't have any real-world counterparts: a database, for example. Essentially, any piece of structured data can be considered an object.

Doing computation with objects

There's one other thing objects have that we haven't talked about yet, and that's *behaviour*. For example, shapes in our hypothetical drawing program could have the ability to draw themselves on screen, or scale themselves to different sizes. That's behaviour: when the object doesn't just passively store data, but *does* something with it.

Behaviour also requires *logic*: in other words, code. While we can store data about an object in its fields, that data is *static*: all we can do is set it and read it back. If we want to do any kind of computation with it, we have to write a function and pass it the object as a parameter.

For example, let's suppose we wanted to extend our `Book` struct to include the book's price. To avoid the loss of precision that we've seen with floating-point values previously in this book, let's represent the price as an integer number of cents:

```
type Book struct {
    Title           string
    Author          string
    Copies          int
    ID              int
    PriceCents      int
    DiscountPercent int
}
```

(Listing 18.1)

And now let's say that we also want to be able to discount certain books, as a special offer. We could represent this with a `DiscountPercent` field on the book. For example, `DiscountPercent: 50` would give 50% off the book's cover price.

To calculate the *net* price of the book (that is, the price after applying the discount), we could imagine some function `NetPriceCents` that takes a `Book` value and returns the discounted price. In fact, we can do more than imagine. Let's write it. We start with a test, of course.

GOAL: Write a test for the `NetPriceCents` function. If you get stuck, review previous chapters, and try to describe the behaviour you want in words, before translating it into Go code. Frame your test in terms of `want` and `got`.

Testing NetPriceCents

Let's start by setting up some book with a given price and discount amount, and then see if we get the expected value of `NetPriceCents` for that book. Here's what that might look like:

```
func TestNetPriceCents(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title: "For the Love of Go",
        PriceCents: 4000,
        DiscountPercent: 25,
    }
    want := 3000
    got := bookstore.NetPriceCents(b)
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing 18.1)

Let's write a null implementation of `NetPriceCents`—just enough to compile, but fail the test:

```
func NetPriceCents(b Book) int {
    return 0
}
```

This should give us a test failure, because 0 is certainly not equal to 3000, and indeed that's what we see:

```
--- FAIL: TestNetPriceCents (0.00s)
```

```
    bookstore_test.go:18: want 3000, got 0
```

GOAL: Write the real implementation of `NetPriceCents`, so that the test passes.

There are many different acceptable ways to write this function. Here's what I came up with:

```
func NetPriceCents(b Book) int {
    saving := b.PriceCents * b.DiscountPercent / 100
    return b.PriceCents - saving
}
```

(Listing 18.1)

If yours looks different, no problem—so long as the test passes.

Methods

We can imagine many other functions like `NetPriceCents` that take a `Book` parameter, do some computation on it, and return the result. In fact, this pattern is so useful that Go gives us a short-cut way to write it:

```
got := b.NetPriceCents()
```

(Listing 18.2)

In other words, if we have some book variable `b`, we can ask for its `NetPriceCents` without *passing* `b` to a function. Instead, we call a *method* on `b` named `NetPriceCents`.

You could think of a method as being like a kind of *dynamic* struct field: every time you ask for its value, it can compute the result on demand.

Defining methods

A method definition in Go looks very similar to a function, but with one small difference. It has a parameter called the *receiver* that represents the object that the method is called on.

Let's see what `NetPriceCents` looks like as a method:

```
func (b Book) NetPriceCents() int {
    saving := b.PriceCents * b.DiscountPercent / 100
    return b.PriceCents - saving
}
```

(Listing 18.2)

Can you spot the difference? Everything inside the function is the same, but it's the *signature* (the `func...` part) that has changed.

The `b Book` parameter has moved *before* the function name, and the function now takes no parameters at all, so the parentheses *after* its name are empty.

To the code inside the function body, the receiver just appears as a normal parameter, so we don't need to change the implementation of `NetPriceCents`, only tweak its signature a little.

GOAL: Update your test to make `NetPriceCents` a method instead of a function, and update the function definition so that the test passes.

It's worth knowing that an object's methods (such as `NetPriceCents`) are closely associated with its type (`Book`) and must be defined in the same package as the type. That means we can't add a method to a type from someone else's package, or from a standard library package, for example.

Methods on non-local types

We've seen how to add methods to a struct type; can we add them to other types of data, too? Let's create a new package so that we can play around with this idea. Create a new folder, wherever you keep your Go code (but not inside the `bookstore` folder), and name it `mytypes`.

Inside this folder, run `go mod init mytypes`, to tell Go you're creating a new module called `mytypes`, and create a new empty file called `mytypes.go`.

What's the first thing we should try to add a method to? How about `int`? Add this code to your `mytypes.go` file, and we'll see what happens:

```
package mytypes

// Twice multiplies its receiver by 2 and returns
// the result.
func (i int) Twice() int {
    return i * 2
}
```

(Listing 19.1)

If you run `go build` in this folder now, you'll find this code doesn't work, because the compiler complains:

```
cannot define new methods on non-local type int
```

This is because `int` is a *non-local* type; that is, it's not defined in our package. Are our attempts to add behaviour to integers completely doomed, though? Not necessarily.

Creating custom types

It's actually quite easy to create new types from existing ones in Go: all you need to do is use the `type` keyword, like this:

```
type MyInt int
```

(Listing 19.2)

That might seem a bit pointless, since it just says that a `MyInt` is an `int`. How does that help us?

Well, we didn't do anything particularly clever with this type definition, but what we *did* do was create a new type that we "own": that is, a *local* type. And that means we can define a method on it:

```
func (i MyInt) Twice() MyInt {
    return i * 2
}
```

(Listing 19.2)

This is perfectly acceptable to Go, since `MyInt` is a *local* type (that is to say, it's defined in this package). We can do whatever we want with it, including adding methods such as `Twice`. We've simply defined `MyInt` in terms of some *underlying type* that already exists: `int`, in this case.

Let's write a test so that we can see how to use such a method. Create the file `mytypes_test.go` and add:

```
package mytypes_test

import (
    "mytypes"
    "testing"
)

func TestTwice(t *testing.T) {
    t.Parallel()
    input := mytypes.MyInt(9)
    want := mytypes.MyInt(18)
    got := input.Twice()
    if want != got {
        t.Errorf("twice %d: want %d, got %d", input, want,
            got)
```

```
    }
}
```

(Listing 19.2)

You might be wondering why we didn't say, for example, just:

```
input := 9
want := 18
```

The reason is that, as you know, Go considers `int` and `MyInt` distinct types. You can't pass an `int` value where a `MyInt` is expected, and vice versa. Since `Twice` is a method on `MyInt`, we can't call it on an `int`. And, similarly, the value returned by `Twice` is a `MyInt`, so we can't compare it with an `int` directly.

What we can do instead is convert some integer literal (`9`) to a `MyInt` value using a *type conversion*:

```
input := mytypes.MyInt(9)
```

We give the name of the type we want (`MyInt`), and then in parentheses the value we'd like to convert (`9`).

Not every type conversion is possible; for example, trying to convert a string to an integer gives a compile error:

```
x := int("Hello")
// cannot convert "Hello" (type untyped string) to type int
```

But in this case it's perfectly possible, and indeed necessary for our test to work.

We hope this test will pass, since the `Twice` code is correct, and indeed it does.

More types and methods

So now we're starting to figure out what we can add methods to in Go. If it's a local type that we defined ourselves, we can give it methods. On the other hand, if the type is defined outside our current package, we can't.

But we can always create a new type based on the type we're interested in, and add a method to *that*. Let's explore this idea a bit further.

For example, let's suppose we want to create a type `MyString`, based on `string`, with a method `Len` that tells us the length of the string. As usual, we'll start with a test.

GOAL: Write a test called `TestMyStringLen`. It should create a value of type `mytypes.MyString`. The test should then call a method `Len` on the value and check the result. If you get stuck, review the `TestTwice` function from earlier in the chapter, and see if you can adapt it.

If you don't want spoilers, look away now. Here's my attempt at this:

```
func TestMyStringLen(t *testing.T) {
    t.Parallel()
    input := mytypes.MyString("Hello, Gophers!")
    want := 15
    got := input.Len()
    if want != got {
        t.Errorf("%q: want len %d, got %d", input, want, got)
    }
}
```

(Listing 19.3)

GOAL: Make the test pass. If you have any trouble, refer to the code for the `MyInt` type and the `Twice` method in Listing 19.2).

The implementation isn't really important here; this is really just about getting the hang of defining and calling methods. Here's what I wrote:

```
// MyString is a custom version of the `string`
// type.
type MyString string

// Len returns the length of the string.
func (s MyString) Len() int {
    return len(s)
}
```

(Listing 19.3)

Creating a Catalog type

Let's return to Happy Fun Books now, and see how to apply these ideas to our bookstore application.

Recall from earlier on that we created a variable to hold our catalog of books. We started off with a slice variable (a `[]Book`), and we wrote a function `GetAllBooks` that returned the slice of all books.

Then, in the chapter introducing maps, we upgraded our catalog variable to a map. It's a map of integer IDs to `Book` elements, so its type is `map[int]Book`. We upgraded our `GetAllBooks` function to deal with the new catalog type.

Now that we've added methods to our Go toolbox, it sounds like it would be useful to have `GetAllBooks` be a method on the catalog, doesn't it? In fact, any function that queries or updates the catalog should probably be a method on the catalog instead.

We know our catalog will be a `map[int]Book`, so suppose we start by trying to turn `GetAllBooks` into a method on this type. As a first attempt, we might write something like the following:

```
func (c map[int]Book) GetAllBooks() []Book {
```

But that's not allowed:

```
invalid receiver type map[int]Book (map[int]Book is not a
defined type)
```

This makes sense, since even though `map[int]Book` is not one of Go's built-in types, it's also not a type that we defined ourselves. As we've seen, we can only define methods on types that we've defined using the `type` keyword.

But now you know how to get around that, don't you? We can create a new local type with some name (let's call it `Catalog`, since that's what it will be used to store), and define it as `map[int]Book`. We should be able to add a method to this new type.

As usual, let's start with a test. Here's our current version of `TestGetAllBooks`:

```
func TestGetAllBooks(t *testing.T) {
    t.Parallel()
    catalog := map[int]bookstore.Book{
        1: {ID: 1, Title: "For the Love of Go"},
        2: {ID: 2, Title: "The Power of Go: Tools"},
    }
    want := []bookstore.Book{
        {ID: 1, Title: "For the Love of Go"},
        {ID: 2, Title: "The Power of Go: Tools"},
    }
    got := bookstore.GetAllBooks(catalog)
    sort.Slice(got, func(i, j int) bool {
        return got[i].ID < got[j].ID
    })
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing 17.3)

We start by setting up the initial catalog, assigning a map literal to the `catalog` variable. Specifically, a `map[int]Book` literal. But we're saying we'll have a special custom type defined for the catalog, named `Catalog`. What would a literal of that type look like?

Well, the same as any other literal, in fact. We write the type name, followed by some data in curly braces:

```
catalog := bookstore.Catalog{
    1: {ID: 1, Title: "For the Love of Go"},
    2: {ID: 2, Title: "The Power of Go: Tools"},
}
```

(Listing 19.4)

That takes care of the catalog setup. Next, we need to update the way we call `GetAllBooks` to reflect the fact that it's now a method on the `catalog` variable:

```
got := catalog.GetAllBooks()
```

(Listing 19.4)

We expect some compile errors now, since we've referred to two things that don't yet exist: the `Catalog` type, and the `GetAllBooks` method on that type.

GOAL: Fix these compile errors, so that the test passes.

Don't worry if you find this challenging at first, because there are a lot of moving parts to sort out. Let's take it step by step. First, we need to define our new type in the `bookstore` package:

```
type Catalog map[int]Book
```

(Listing 19.4)

Next, we'll make the necessary changes to `GetAllBooks` to convert it to a method on `Catalog`:

```
func (c Catalog) GetAllBooks() []Book {
    result := []Book{}
    for _, b := range c {
        result = append(result, b)
    }
    return result
}
```

(Listing 19.4)

There's no change needed to the actual logic of the function; the only thing that's different is the signature of `GetAllBooks`. It now has a *receiver* (as all methods must), of type `Catalog`, and it needs a name so that we can refer to it within the function. `c` for "catalog" sounds reasonable (receiver names are conventionally quite short, because we can assume we'll be referring to them often within the body of the method).

This passes our updated test, so we've successfully defined the new `Catalog` type along with the `GetAllBooks` method.

Can you extend this idea to `GetBook`, too?

GOAL: Update the `GetBook` function to be a method on the catalog. Start by modifying the existing tests on `GetBook` for both valid and invalid input. Make sure you get the expected compile errors, and then make the necessary changes to `GetBook` so that the test compiles and passes.

If you get stuck, have a look at my version of the test and the implementation in Listing 19.5.

Takeaways

- A Go struct can not only store data in its fields, but also define *behaviour* with its *methods*.
- A method is equivalent to a function, and works the same way, but it singles out one of its parameters—the *receiver*—as special: in some sense, the method is *about* this receiver, and it's part of the definition of the receiver's type.
- We call a method using the dot notation: `b.NetPriceCents()`.
- Because a method is part of a type definition, it follows that we can't write methods on types defined outside the current package, and that includes built-in types such as `int`.
- But we can trivially create new types based on existing types (for example, `type MyInt int`): then we can add a method on our new type.
- However, a new type such as `MyInt` is distinct from its *underlying type*, so we need to explicitly convert values to the new type using a *type conversion*.

9. Wrapper's delight

You saw in the previous chapter that you can create a new type based on some existing type, just by writing a type definition like:

```
type MyInt int
```

(Listing 19.2)

This enables you to add methods to `MyInt`, which you can't do with the `int` type, because you don't own it. But what about types *that already have methods*?

The `strings.Builder`

There's a very useful standard library type called `strings.Builder` that allows you to efficiently build up one big string from lots of little ones. Let's have a play with this type first, and then we'll try to extend it with some new methods.

In your `mytypes` package, start by adding a test that will demonstrate how `strings.Builder` works:

```
func TestStringsBuilder(t *testing.T) {
    t.Parallel()
    var sb strings.Builder
    sb.WriteString("Hello, ")
    sb.WriteString("Gophers!")
```

```
    want := "Hello, Gophers!"
    got := sb.String()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
    wantLen := 15
    gotLen := sb.Len()
    if wantLen != gotLen {
        t.Errorf("%q: want len %d, got %d", sb.String(),
            wantLen, gotLen)
    }
}
```

(Listing 20.1)

Let's break this down step by step. We don't need to do anything special to create a new `strings.Builder`; declaring a variable `sb` of that type is enough:

```
var sb strings.Builder
```

(Listing 20.1)

We can now write a couple of string fragments to the builder:

```
sb.WriteString("Hello, ")
sb.WriteString("Gophers!")
```

(Listing 20.1)

Then we can retrieve the entire contents of the builder by calling its `String` method, and we expect it to be the sum of the fragments we've written so far:

```
want := "Hello, Gophers!"
got := sb.String()
```

(Listing 20.1)

To get the length of the current contents, we call the builder's `Len` method. In this case, we expect the length to be 15, which is the number of characters in the string `"Hello, Gophers!"`.

```
wantLen := 15
gotLen := sb.Len()
```

(Listing 20.1)

Creating a type based on `strings.Builder`

Let's try defining a new type based on `strings.Builder` and see what we can do with it. Suppose we call it `MyBuilder`, for example:

```
type MyBuilder strings.Builder
```

This raises an interesting question. Will `MyBuilder` have the same methods as `strings.Builder`? Let's see what happens if we write a test that calls the `Len` method on a `MyBuilder` value, for example:

```
func TestMyBuilderLen(t *testing.T) {
    t.Parallel()
    var mb mytypes.MyBuilder
    want := 15
    got := mb.Len()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

(Listing 20.1)

Unfortunately, this doesn't compile:

```
mb.Len undefined (type mytypes.MyBuilder has no field or
    method Len)
```

It looks like we don't get `strings.Builder`'s methods when we define a new type based on it. That's a shame. But all is not lost: because we defined `MyBuilder` ourselves, we can add our *own* methods to it.

For example, suppose we wanted to add some method `Hello`, that just returns the fixed string `Hello, Gophers!`. Not super useful, perhaps, but we're just experimenting. Let's write a test first:

```
func TestMyBuilderHello(t *testing.T) {
    t.Parallel()
    var mb mytypes.MyBuilder
    want := "Hello, Gophers!"
    got := mb.Hello()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing 20.1)

Over to you now to implement the method itself!

GOAL: Make this test pass. If you run into trouble, refer back to the previous chapter and the `MyInt` type, with its `Twice` method.

Here's my version:

```
type MyBuilder strings.Builder

func (mb MyBuilder) Hello() string {
    return "Hello, Gophers!"
}
```

(Listing 20.1)

Wrapping `strings.Builder` with a struct

So, now we know how to create a new type based on an existing type, and define our own methods on it. But what if we *also* want our new type to have the methods of the underlying type? For example, what if we wanted `MyBuilder` to have the `Len` and `String` methods that `strings.Builder` has?

Well, we can't do that directly, but there *is* a way to do something similar. We can create a *struct* type, with a field of type `strings.Builder`. This is sometimes called *wrapping* `strings.Builder`.

Let's see what that would look like by modifying our test. Suppose we name our struct field `Contents`, and we assume that we'll be able to call the usual `strings.Builder` methods on it such as `WriteString`, `String`, and `Len`.

```
func TestMyBuilder(t *testing.T) {
    t.Parallel()
    var mb mytypes.MyBuilder
    mb.Contents.WriteString("Hello, ")
    mb.Contents.WriteString("Gophers!")
    want := "Hello, Gophers!"
    got := mb.Contents.String()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
    wantLen := 15
    gotLen := mb.Contents.Len()
    if wantLen != gotLen {
        t.Errorf("%q: want len %d, got %d",
```

```
            mb.Contents.String(), wantLen, gotLen)
    }
}
```

(Listing 20.2)

We expect some compilation errors here, since we haven't yet updated the `MyBuilder` type definition, and the compiler doesn't disappoint us:

```
mb.Contents undefined (type mytypes.MyBuilder has no field or
method Contents)
```

Can you figure out what to do?

GOAL: Make this test pass.

One of the nice things about developing software guided by tests is that, having done most of the hard thinking in the test itself, we usually don't find it too difficult to write the necessary implementation. It's merely a matter of cranking the handle, if you like, and following the logic of what the test demands.

In this case, we know that `MyBuilder` needs to be a struct type, because it has a field. We also know what the name of that field must be, because it's referenced in the test as `Contents`.

Finally, we know the type of that field, because it needs to have the methods of `strings.Builder`:

```
type MyBuilder struct {
    Contents strings.Builder
}
```

(Listing 20.2)

There's nothing magical about the way this works: in the test, `mb.Contents` is a `strings.Builder`, so it has the methods that a `strings.Builder` has. But because `MyBuilder` is a local type, defined by us, we can also add any methods to it that we want, such as `Hello`.

You can see how to use this idea to extend any non-local type that has methods, can't you?

GOAL: Write a test for a new type in the `mytypes` package named `StringUppercaser`. This type wraps `strings.Builder`, and adds a `ToUpper` method that returns the contents of the builder with all characters converted to uppercase. Implement everything necessary to make the test pass. (Hint: use the standard library function `strings.ToUpper` to do the actual case conversion.)

Here's my attempt at the test:

```
func TestStringUppercaser(t *testing.T) {
    t.Parallel()
```

```
    var su mytypes.StringUppercaser
    su.Contents.WriteString("Hello, Gophers!")
    want := "HELLO, GOPHERS!"
    got := su.ToUpper()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing 20.3)

I'm assuming that there'll be a `Contents` field on my struct type, as before, that I can call `WriteString` on, but I'm also assuming, per the instructions, that the struct type will have a `ToUpper` method. Here's how I've implemented that:

```
// StringUppercaser wraps strings.Builder.
type StringUppercaser struct {
    Contents strings.Builder
}

func (su StringUppercaser) ToUpper() string {
    return strings.ToUpper(su.Contents.String())
}
```

(Listing 20.3)

There's not much code here, so it's tempting to think that nothing interesting is going on. When you can confidently solve a challenge such as this, though, it shows that you've fully understood some important rules about Go types and methods—perhaps more deeply than many working Go programmers. If you get stuck at any point, review the last couple of chapters and try to find the missing step.

A puzzle about function parameters

By now, you know a bit about functions in Go, though we haven't talked about them in great detail. For example, you know they can take *parameters*, which is to say that when you call a function, you can pass some values to it.

Let's look a little more closely at how this works. Suppose we write a test for a function in the `mytypes` package called `Double`. It should take one parameter, an integer, and multiply it by 2.

Here's something that should do the job:

```
func TestDouble(t *testing.T) {
```

```
    t.Parallel()
    var x int = 12
    want := 24
    mytypes.Double(x)
    if want != x {
        t.Errorf("want %d, got %d", want, x)
    }
}
```

(Listing 21.1)

Nothing fancy here: `Double` is a function, not a method, and all it needs to do is multiply its input by 2. Here's my first attempt at an implementation:

```
func Double(input int) {
    input *= 2
}
```

(Listing 21.1)

Recall from our work in an earlier chapter on the Buy function that Go lets us write `input *= 2` as a short form for `input = input * 2`.

So does the test pass now? Well, slightly surprisingly, no:

```
--- FAIL: TestDouble (0.00s)
    mytypes_test.go:14: want 24, got 12
```

What's going on? Is the `Double` function broken?

Actually, `Double` is doing exactly what we asked it to do. It receives a parameter we call `input`, and it multiplies that value by 2:

```
input *= 2
```

So why is it that, in the test, when we set `x` to 12, and call `Double(x)`, the value of `x` remains 12? Shouldn't it now be 24?

```
var x int = 12
want := 24
mytypes.Double(x)
```

Parameters are passed by value

The answer to this puzzle lies in what happens when we pass a value as a parameter to a Go function. It's tempting to think, if we have some variable `x`, and we call `Double(x)`, that the `input` parameter inside `Double` is the variable `x`. But that's not the case.

In fact, the value of `input` will be the same as the value of `x`, but they're two independent variables. The `Double` function can modify its local `input` variable as much as it likes, but that will have no effect on the `x` variable back in the test function—as we've just proved.

The technical name for this way of passing function parameters is *pass by value*, because `Double` receives only the value of `x`, not the original `x` variable itself. This explains why `x` wasn't modified in the test.

Creating a pointer

Is there any way, then, to write a function that can modify a variable we pass to it? For example, we'd like to write a version of `Double` that will actually have an effect on `x` when we call `Double(x)`. Instead of just taking a copy of the value of `x` at the moment of the function call, we want to pass `Double` some kind of reference to `x`. That way, `Double` could modify the original `x` directly.

Go lets us do exactly that. We can create what's called a *pointer* to `x`, using this syntax:

```
mytypes.Double(&x)
```

(Listing 21.2)

You can think of the `&` here as the *sharing operator*; it lets you share a variable with the function you're passing it to, so that the function can modify it.

Declaring pointer parameters

There's still something missing, though, because our modified test doesn't compile:

```
cannot use &x (type *int) as type int in argument to
mytypes.Double
```

The compiler is saying that `Double` takes a parameter of type `int`, but what we tried to pass it was a value of type `*int` (pronounced "pointer to int").

These are two distinct types, and we can't mix them. If we want the function to be able to take a `*int`, we'll need to update its signature accordingly:

```
func Double(input *int) {
```

(Listing 21.2)

It's worth adding that the type here is not just "pointer", but specifically "pointer to int". For example, if we tried to pass a `*float64` here, that wouldn't work. A `*float64` and a `*int` are both pointers, but since they're pointers to different *base* types, they are also different from each other.

What can we do with pointers?

We're still not quite done, because even with our updated function signature, the compiler isn't happy with this line:

```
input *= 2
```

It complains:

```
invalid operation: input *= 2 (mismatched types *int and int)
```

Another type mismatch. It's saying "you tried to multiply two different kinds of thing". The numeric constant literal 2 is interpreted as an `int`, while `input` is a pointer.

The * operator

Instead of trying to do math with the pointer itself, we need the value that the pointer *points to*. Because a pointer is a reference to some variable, the fancy name for this is *dereferencing* the pointer.

To get the value pointed to by `input`, we write `*input` ("star-input"):

```
*input *= 2
```

(Listing 21.2)

Nil desperandum

You know that every data type in Go has some default value. If you declare a variable of type `int`, for example, it automatically has the value 0 unless you assign some other value to it.

So what's the default value of a pointer type? If you declare a variable of type `*int`, what value does it have?

The answer is the special value `nil`, which you've encountered many times already in connection with `error` values. Just as a nil error value means "no error", a nil pointer value means "doesn't point to anything".

It makes no sense to dereference a nil pointer, then, and if this situation arises while your program is running, Go will stop execution and give you a message like:

```
panic: runtime error: invalid memory address or nil pointer
dereference
```

This shouldn't happen under normal circumstances, so "panic" in this context means something like "unrecoverable internal program error".

Pointer methods

We know that a function parameter can be a pointer, so can a method receiver be a pointer, too? Certainly. Such a method is called a *pointer method*.

Why would a method take a pointer receiver? Only because, just as in the function case, it needs to *modify* that object. In other words, if we want to write a method that modifies its receiver, then that receiver must be a pointer type.

We wanted the `Double` function to modify the value passed to it, and we achieved that using a pointer. Could we turn `Double` into a method that takes a pointer receiver, then? Let's find out.

We already know we can't create a method on `int`, but we also know how to deal with that: create a new type `MyInt` instead, and define the method on that.

We'll start by modifying the test to create a pointer to a `MyInt` value, and then call the `Double` method on it:

```
func TestDouble(t *testing.T) {
    t.Parallel()
    x := mytypes.MyInt(12)
    want := mytypes.MyInt(24)
    p := &x
    p.Double()
    if want != x {
        t.Errorf("want %d, got %d", want, x)
    }
}
```

(Listing 21.3)

GOAL: Update the definition of `Double` to make this test pass. You know how to write methods, and you also now know how to declare pointer parameters. Can you combine these two ideas to solve the problem?

Here's one solution:

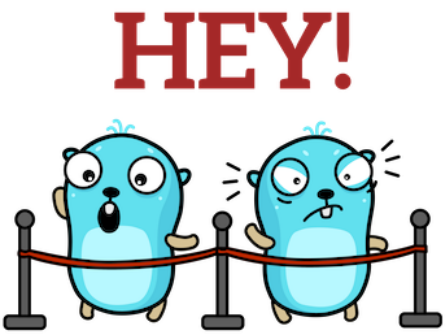
```
type MyInt int

func (input *MyInt) Double() {
    *input *= 2
}
```

(Listing 21.3)

Takeaways

- When you define a new type, it doesn't automatically "inherit" any methods that the underlying type has.
- But you can define a *struct* type instead, containing a field of the type that you want to *wrap* with new methods.
- When you pass a variable to a function, the function actually receives a *copy* of that variable's value.
- So if the function modifies the value, that change won't be reflected in the original variable.
- When you *want* the function to modify the original variable, you can create a *pointer* to the variable, and pass that instead.
- A pointer gives *permission to modify* the thing it points to.
- The `&` operator creates a pointer, so `&x` gives a pointer to `x`.
- A function that takes a pointer must declare the appropriate parameter type: `*int` for "pointer to `int`", for example.
- Just as ordinary types are distinct and non-interchangeable, so are pointers to those types: for example, `*int` and `*float64` are different types.
- The only thing we're allowed to do with a pointer value is *dereference* it using the `*` operator to get the value it points to, so `*p` gives the value of whatever variable `p` points to.
- The default (zero) value of any pointer type is `nil`, meaning "doesn't point to anything".
- Trying to dereference a pointer that happens to be `nil` will cause the program to *panic* and terminate with an error message.
- Just as functions can take pointer parameters, so methods can take *pointer receivers*, and indeed they must do so if they want to modify the receiver.

10. Very valid values



Now that you know how to use pointer methods to modify the variables they're called on, let's see how to apply that to the valuable work you're doing at Happy Fun Books.

At the moment, if you want to modify some information about a book, you'd update the relevant field on the `Book` struct directly with a new value.

But there's a potential problem here: what if you make a mistake and assign some value that doesn't make sense? For example, setting a negative `PriceCents` on a book, or a `DiscountPercent` that's greater than 100, or less than zero.

At the moment, we've no protection against that kind of problem: we don't have any *validation* of the changes we make to books. Let's think about how to solve this.

Validating methods

A wise bookseller knows how to adjust prices to keep up with demand. So we'll need to be able to update prices on the books at Happy Fun Books, and we don't want to modify the `PriceCents` field directly, because there's no way to validate changes like that.

Instead, suppose we created a method to make the change for us. Let's call it `SetPriceCents`. We can pass it the new price as a parameter and, importantly, it'll be able to check the price to ensure it's not negative.

GOAL: Write suitable tests for a `SetPriceCents` method on the `Book` type. You'll need at least two tests, because we're saying there are two different behaviours: one for valid

input, and another for invalid input. If you have trouble, refer back to previous tests you've written for functions that validate their input, such as `calculator.Divide` and `bookstore.GetBook`.

Let's consider the valid input case first. We'll need some `Book` value to operate on, so let's start by defining that:

```
func TestSetPriceCents(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title: "For the Love of Go",
        PriceCents: 4000,
    }
```

(Listing 22.1)

We'll be calling the `SetPriceCents` method on this value with some new price (let's say 3000), and that's the price we'll `want` it to be afterwards:

```
want := 3000
err := b.SetPriceCents(want)
if err != nil {
    t.Fatal(err)
}
```

(Listing 22.1)

Since this is the valid input case, we don't expect an error here, so the test should fail if we see one. Finally, we'll compare `want` and `got` in the usual way.

```
got := b.PriceCents
if want != got {
    t.Errorf("want updated price %d, got %d", want, got)
}
```

(Listing 22.1)

GOAL: Make this test pass.

Since we don't need to actually do any validation yet, this should be pretty straightforward. We can define a method `SetPriceCents` on the `Book` type. Let's try:

```
func (b Book) SetPriceCents(price int) error {
    b.PriceCents = price // nope
    return nil
}
```

This looks reasonable, but there's a problem:

```
--- FAIL: TestSetPriceCents (0.00s)
    bookstore_test.go:21: want updated price 3000, got 4000
```

Oops! What's gone wrong? Can you figure it out?

Recall from the previous chapter introducing pointers that if a method is supposed to modify its receiver, then the receiver must be a pointer type (for example, `*Book`).

If instead we accidentally wrote a *value method* (one that takes a non-pointer receiver, such as `Book`), then any changes to it we might make in the method don't persist. This can lead to subtle bugs that are hard to detect without careful testing.

In fact, the `staticcheck` linter (Visual Studio Code and some other Go editors run this for you automatically) will warn you about this problem:

```
ineffective assignment to field Book.PriceCents (SA4005)
```

If you're not sure you understand the issue, or how to solve it, review the chapter on pointers. Don't feel bad if you're having trouble with this concept: it's one that trips up many, many people learning Go for the first time.

The solution is to do just what we did with the `Double` method in the previous chapter: modify it to take a pointer receiver instead of a value.

```
func (b *Book) SetPriceCents(price int) error {
```

Automatic dereferencing

In the previous chapter, you learned that when a function (or a method) wants to do something with a pointer, it needs to dereference it first, using the `*` operator, to get the value that it points to. How would we do that here?

Actually, we don't need to do anything at all! We can write, exactly as we did before:

```
b.PriceCents = price
```

Since `b` is now a pointer (specifically, a `*Book`), how does this work? Well, Go knows that pointers don't have fields, only structs do. So if you write something like `b.PriceCents` that looks like a struct field, Go is smart enough to assume you meant "dereference `b` to get the struct it points to, then modify its `PriceCents` field".

This is called *automatic dereferencing*, and since most methods on structs are pointer methods, it's very helpful. We would otherwise have to write something like this, which feels awkward:

```
(*b).PriceCents = price
```

With that in mind, we can try the test again to see if the change to `PriceCents` has persisted, now that we're using a pointer:

```
PASS
```

Great! That's the valid input case taken care of. We can go on to write the test for the invalid input case.

```
func TestSetPriceCentsInvalid(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title: "For the Love of Go",
        PriceCents: 4000,
    }
    err := b.SetPriceCents(-1)
    if err == nil {
        t.Fatal("want error setting invalid price -1, got nil")
    }
}
```

(Listing 22.1)

We try to set some negative price on the book (-1 will do just fine), and we fail the test unless `SetPriceCents` returns error. And that's all this test needs to do.

It doesn't pass yet, because we haven't done the work, but let's just see what that failure looks like:

```
want error setting invalid price -1, got nil
```

Great! Now we can go ahead and make it pass.

GOAL: Add validation to `SetPriceCents` so that the test passes.

Here's one way to do it:

```
func (b *Book) SetPriceCents(price int) error {
    if price < 0 {
        return fmt.Errorf("negative price %d", price)
    }
    b.PriceCents = price
    return nil
}
```

(Listing 22.1)

Always valid fields

This is a big improvement, because now if some poor harassed programmer mistakenly calls `SetPriceCents` with an invalid price, the program will detect the mistake automatically and alert them to it.

But what's to stop someone from just setting the `PriceCents` field directly, without going via the `SetPriceCents` method that you so obligingly provided for them? Well, nothing. It's an exported field on the `Book` struct, so anyone who has a `Book` value can set the field to any value they want, including negative ones.

Is there anything we can do about this? Maybe there is. Books are usually arranged in bookstores by *category*, such as romance novels, programming books, or junior wizard school stories. So let's add a new field to our `Book` struct that identifies the category the book belongs to, and we'll try to figure out a way to make sure that it can only hold valid values.

What's valid? Well, in this case it would have to be one of a predefined set of values, representing the categories we have in the Happy Fun Books store: `"Romance"`, `"Western"`, or `"Quantum Mechanics"`, for example.

Adding a new `string` field to the struct is no problem, but we want to make it impossible for people to assign arbitrary values to that field. One way would be to make the field unexported (`category`), and then it won't be possible to access this field directly outside the `bookstore` package.

People will still need to be able to set and read that field, naturally, but we can give them a way to do that using methods. For example, they could read the category by calling a method `Category` on the book, and set it by calling `SetCategory`. These are sometimes called *accessor methods*, since they give users a way to access a field that they otherwise couldn't.

This is exactly the same idea as with `SetPriceCents`, except that we're making the relevant field unexported by giving it a lowercase name. See if you can write the appropriate tests yourself before reading on to see how I'd approach this.

GOAL: Write tests for a `SetCategory` method on `Book`, including both valid and invalid input cases. For the moment, let's assume that we only have one valid category, `"Autobiography"`, and that trying to set any other category would cause an error.

Let's consider the invalid input case first this time, because it's usually easier. We'll start with a book:

```
b := bookstore.Book{
    Title: "For the Love of Go",
}
```

(Listing 22.2)

We'll try to set some non-existent category on it (that is, any category other than `"Autobiography"`), and fail the test unless there's an error:

```
err := b.SetCategory("bogus")
if err == nil {
    t.Fatal("want error for invalid category, got nil")
}
```

(Listing 22.2)

The valid input case starts out similarly, except that it *doesn't* expect an error:

```
b := bookstore.Book{
    Title: "For the Love of Go",
}
want := "Autobiography"
err := b.SetCategory(want)
if err != nil {
    t.Fatal(err)
}
```

(Listing 22.2)

It should also check that the category was actually updated (in case we made a mistake like the one I made earlier in this chapter, by omitting to take a pointer receiver). But there's a problem: we're saying that the `category` field will be unexported, meaning that we can't refer to it from outside the `bookstore` package. How, then, can the test check its value?

Well, we already decided that we'll need a `Category` method too, so that users can read as well as set the category. And it turns out we need it ourselves, for the purpose of testing the `SetCategory` method! This is actually very common when writing accessor methods: you need to both set *and* get the value in a test. So let's do just that:

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

(Listing 22.2)

Do we need a separate test for `Category`? Not really, though there'd be no harm in writing one if you want to. If `SetCategory` doesn't work for some reason, then we'll detect that by calling `Category`. On the other hand, if `Category` doesn't work, then we won't see the category that we just set with `SetCategory`. And if *neither* of them works, the test will still fail, because the book just won't have a category at all.

So a single pair of tests, for valid and invalid inputs, exercises the behaviour of both `Category` and `SetCategory`. Great. Let's go ahead and write these methods now.

GOAL: Implement `Category` and `SetCategory` so that the tests pass.

This might be a little bit of a challenge, but as usual, you have all the information you need, and if you get stuck, you can review the material in this chapter so far and it will help you.

We can start by adding the necessary field to our struct type:

```
type Book struct {
    ... // existing fields omitted
    category string
}
```

(Listing 22.2)

Note that the field name `category` starts with a lowercase letter, making it unexported. That's the crucial bit of syntax we need to enforce the "always valid field" idea. If you find adding this field causes your existing tests to fail, see the "Unexported fields and `cmp.Equal`" section that follows this one.

The implementation of `SetCategory` is pretty similar to the `SetPriceCents` method you wrote earlier in this chapter:

```
func (b *Book) SetCategory(category string) error {
    if category != "Autobiography" {
        return fmt.Errorf("unknown category %q", category)
    }
    b.category = category
    return nil
}
```

(Listing 22.2)

I'm sure you remembered to have `SetCategory` take a pointer receiver instead of a value (and if not, the test will remind you, as will `staticcheck`).

The `Category` method is even simpler:

```
func (b Book) Category() string {
    return b.category
}
```

(Listing 22.2)

Because `Category` isn't supposed to *modify* the receiver, it doesn't need to take a pointer. (It wouldn't do any harm if it did, but it's just not necessary.)

And we've done what we set out to do: create an *always valid field*. No matter what kind of `Book` value someone has, or what they do to it, its category can never be set to something invalid.

If they try to call `SetCategory` with invalid input, they'll get an error: your test proves that. If they try to write directly to the `category` field, their program won't compile. Making it impossible to compile incorrect programs is the best kind of validation!

Unexported fields and `cmp.Equal`

If you're adding each new test and method to your `bookstore` project as you work through the book, you might have suddenly encountered a rather alarming error when running tests after adding the `category` field. Something like this:

```
--- FAIL: TestGetAllBooks (0.00s)
panic: cannot handle unexported field at
    {[]bookstore.Book}[1].category: "bookstore".Book
consider using a custom Comparer; if you control the
implementation of type, you can also consider using an
Exporter, AllowUnexported, or cmpopts.IgnoreUnexported
[recovered]
```

Where did that come from? Well, recall that we used `cmp.Equal` in some of our tests, to compare some `Book` struct values. But `cmp.Equal` can't compare structs with unexported fields, because (like anyone else outside the `bookstore` package) it's not allowed to refer to them. That's why it says "cannot handle unexported field".

Since it's quite common in Go to define types with unexported fields, and `cmp.Equal` is invaluable in tests, you're bound to run into this problem at some point, so let's see how to solve it.

In fact, the panic message itself contains the answer: "consider using... cmpopts.IgnoreUnexported". There are various options you can pass to `cmp.Equal` to control how it compares things, and `IgnoreUnexported` is the one we need in this case.

Add the following import to your package:

```
"github.com/google/go-cmp/cmp/cmpopts"
```

Modify the calls to `cmp.Equal` in your tests to change them from this:

```
if !cmp.Equal(want, got) {
```

to this:

```
if !cmp.Equal(want, got,
    cmpopts.IgnoreUnexported(bookstore.Book{})) {
```

This tells `cmp.Equal` to ignore any unexported fields on the `bookstore.Book` struct, preventing the unpleasant panic message.

Always valid structs

Throughout this book we've been thinking about how to represent data in Go, and what we can do with it. For example, we can add behaviour to it, by defining methods on

types. We've also seen how to *constrain* data to allowed values, using unexported fields and validating accessor methods.

We can extend the idea of an "always valid" variable or field to an entire data type. Let's explore that possibility by thinking about something else we'll need in our bookstore application: credit card information.

Happy Fun Books needs to be able to take payments, like any store, and that means we'll need to deal with things like credit cards. Although in practice we will probably use a third-party payment processor to handle the actual card transactions, we might still want to be able to store customers' credit card details to save them time at the checkout, for example.

Suppose we're writing a package to deal with credit cards, then, and we want to make sure that only valid card values are created. Just as you can ensure a field is always valid by making it unexported, could we use the same idea to make an *always valid type*?

We could make an unexported type `card`, for example, that can only hold valid credit card details, and enforce that rule by requiring users to call a *constructor*: a function that creates valid values of our chosen type.

First, have a go yourself to see if you can figure out how to put the necessary pieces together. Then we'll work through it step by step.

GOAL: Create a new module `creditcard`, and write tests for a constructor function `New` that creates values of type `creditcard.card`. This type should have a string field `number` that is guaranteed not to be empty. Make the tests pass.

As usual with any validating function, we'll need to write at least two tests: one for valid input, and one for invalid input. Here's my attempt at the "valid input" test:

```
package creditcard_test

import (
    "creditcard"
    "testing"
)

func TestNew(t *testing.T) {
    t.Parallel()
    want := "1234567890"
    cc, err := creditcard.New(want)
    if err != nil {
        t.Fatal(err)
    }
    got := cc.Number()
```

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

(Listing 23.1)

The behaviour I want is something like "calling `New` with a valid credit card number should return no error, and a card value containing that number". The "no error" part is straightforward, but how can I tell if the value returned by `New` has the number I passed in? I can't inspect the `number` field, because that's unexported. (It has to be, doesn't it? If it were exported, users would be able to directly set any value they liked, bypassing our validation.)

So, just as with the `Book.category` field, we'll need some *getter* method (`Number`) that can retrieve the number for inspection. We won't need a corresponding `SetNumber` method, because `New` already does that, and it doesn't make sense to change the number on an existing card—we'd create a new card instead.

Why a `string` field for the credit card number, rather than `int`, for example? Well, this data is likely to come from user input, such as a web form, so we can make no assumptions about what it contains. They might have entered a letter instead of a digit by mistake. That's one of the things we might like to eventually validate about it.

Implementing the minimum necessary to make this pass looks something like:

```
package creditcard

type card struct {
    number string
}

func New(number string) (card, error) {
    return card{number}, nil
}

func (c *card) Number() string {
    return c.number
}
```

We don't have any validation yet, so let's add the test for that:

```
func TestNewInvalidReturnsError(t *testing.T) {
    t.Parallel()
    _, err := creditcard.New("")
    if err == nil {
```

```
        t.Fatal("want error for invalid card number, got nil")
    }
}
```

(Listing 23.1)

This fails as expected:

```
want error for invalid card number, got nil
```

So let's go ahead and make it pass:

```
func New(number string) (card, error) {
    if number == "" {
        return card{}, errors.New("number must not be empty")
    }
    return card{number}, nil
}
```

(Listing 23.1)

There's nothing especially new to you here, except for the fact that our core struct type is unexported. But you might be puzzled about this, because you know that making an identifier unexported means it can't be used outside the package where it's defined. So how can we use it in the tests?

The answer is that we don't! We never refer to the name `creditcard.card` in any test code. We don't need to. `creditcard.New` returns a value of some type that we don't need to know the name of, because we don't use its name. We call the `Number` method on it. Since the *method*'s name is exported, we're allowed to call it.

This very neatly ensures that values of type `card` are always valid. It's impossible to construct an invalid one, at least from outside the `creditcard` package (try it yourself!)

A set of valid values

Now that we've seen how to use Go's type machinery to ensure that data is validated, let's think about that validation process in a little more detail. So far we've only done very simple validation, such as ensuring a value isn't negative or empty. Can we do more?

We would often like to ensure that some value is one of a predefined set of possibilities, like the book category in the previous chapter. Actually, in that case the set of valid values had only one element, `"Autobiography"`. What if that set were larger? A real bookstore might have dozens of categories, and we feel instinctively that we should be able to do better than writing a bunch of `if` statements.

What would be very convenient is if we had some kind of data structure containing a set of unique elements. Actually, that sounds familiar, doesn't it? The keys of a Go map have exactly those properties, so let's see if we can use a map to store and validate our categories. Suppose we want there to be exactly three valid categories:

```
Autobiography
Large Print Romance
Particle Physics
```

Now this sounds like my kind of bookstore!

Using a map to represent a set

Here's one way we could model the set of categories as a map. We could define a package-level variable in `bookstore.go` that looks something like this:

```
var validCategory = map[string]bool{
    "Autobiography":       true,
    "Large Print Romance": true,
    "Particle Physics":    true,
}
```

Checking if a given value is in this set is as simple as performing a map lookup:

```
if validCategory[category] {
```

If the category is in the map, the result will be `true`, since all the elements of the map are `true`. But what about categories that aren't in the map? As you might remember from the chapter introducing maps, they return the *zero value* for missing keys, and the zero value for `bool` is `false`. So that means invalid categories will return `false` without us having to do anything. Ideal!

By the way, you might have wondered why I named the map `validCategory`: wouldn't `validCategories` make more sense? The answer is that it makes the code that *uses* the map read more naturally. `if validCategory...` reads almost like English, whereas `if validCategories...` just sounds a bit wrong. Little details like this can go a long way to making your code easier and more enjoyable to read.

A question might have occurred to you: Can we implement categories in such a way that the *compiler* can validate them for us? In other words, these names (`"Large Print Romance"` and so on) are currently just arbitrary strings, but could we make them Go identifiers, like variable or function names, so that we can define a set of them in our package?

Go has a special feature that's intended for just this purpose: *constants*.

Defining constants

We define a constant using the `const` keyword:

```
const CategoryAutobiography = "Autobiography"
```

Just as with imports or variables, we can group multiple constant definitions together using parentheses:

```
const (
    CategoryAutobiography = "Autobiography"
    CategoryLargePrintRomance = "Large Print Romance"
    CategoryParticlePhysics = "Particle Physics"
)
```

It's easy to update our `validCategory` map to use these new constants instead of the string literals we had before:

```
var validCategory = map[string]bool{
    CategoryAutobiography:     true,
    CategoryLargePrintRomance: true,
    CategoryParticlePhysics:   true,
}
```

Referring to constants

The real benefit of this is that now the user of our package can refer to these predefined constants, instead of arbitrary strings:

```
b := bookstore.Book{Title: "A Dinner of Onions"}
err := b.SetCategory(bookstore.CategoryLargePrintRomance)
```

Notice that if they should misspell the name of the category, the compiler will pick this up:

```
err := b.SetCategory(bookstore.CategoryLargePrintBromance)
// undefined: bookstore.CategoryLargePrintBromance
```

Standard library constants

This technique is used a lot in the standard library. For example, the `net/http` package defines named constants for all the various HTTP status codes, such as `http.StatusOK`, or `http.StatusNotFound`. Of course, this helps you catch your own mistakes when writing code that deals with HTTP statuses, since if you try to refer to a non-existent constant (`http.StatusBogus`), the compiler won't let you.

Using a constant like `http.StatusOK` also makes your code more readable, since anyone looking at it can see right away what that value signifies, unlike a literal value like 200, which takes a little more thought.

When the values don't matter

If we're just defining a bunch of constants, and the only thing we care about is that some arbitrary value is one of those constants, then the actual *value* of the constant doesn't matter, does it? It could be anything. It doesn't even need to be a string.

Suppose we make it an integer, for example:

```
const (
    CategoryAutobiography = 0
    CategoryLargePrintRomance = 1
    CategoryParticlePhysics = 2
)
```

This would mean that a method like `SetCategory` would now take, not a `string` parameter, but `int`. That sounds a little bit strange: what have integers to do with book categories? Let's trivially define a type `Category` to use instead:

```
type Category int
```

Now we can declare our constants to be of this type:

```
const (
    CategoryAutobiography Category = 0
    CategoryLargePrintRomance = 1
    CategoryParticlePhysics = 2
)
```

Note that it's not necessary to add the type `Category` for each constant, just the first one; all the others will automatically be the same type.

Introducing `iota`

We can imagine extending this idea to any number of categories, just by defining more constants with increasing integer values.

In fact, that's so common in Go, there's a special "magic" constant to do it for us, called `iota`:

```
const (
    CategoryAutobiography Category = iota
    CategoryLargePrintRomance
```

```
    CategoryParticlePhysics
)
```

(Listing 23.2)

When you declare a constant and assign it `iota`, it will get the value 0, and successive constants will get successively increasing values: 1, 2, 3... The word "iota" is the name of the smallest letter in the Greek alphabet, incidentally, and it's come to mean in common usage "the smallest possible amount" of something, as in "not one iota".

The `iota` constant is most useful when we want to declare a set of constants to represent the allowed values of something, but we don't actually care what those values *are*. Instead of making up arbitrary values, we can use `iota` to enumerate them for us. All that we care about is that they're different from each other, and `iota` will make sure of that.

It might be, for example, that we later want to insert new constants somewhere in the middle of the list, and we won't have to manually renumber dozens of them. `iota` renumbers them automagically. Thanks, `iota`!

GOAL: Update your `bookstore` package, including the tests, to define a set of `iota`-enumerated category constants, and modify `SetCategory` to validate the category using a map of these constants.

Let's start with `TestSetCategory`. We can't just check that `CategoryAutobiography` is a valid category anymore, because we've added some new valid categories. Although there are only three at the moment, there could easily be a lot more, so we'll use a table test to make the code easier to read:

```
func TestSetCategory(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title: "For the Love of Go",
    }
    cats := []bookstore.Category{
        bookstore.CategoryAutobiography,
        bookstore.CategoryLargePrintRomance,
        bookstore.CategoryParticlePhysics,
    }
    for _, cat := range cats {
        err := b.SetCategory(cat)
        if err != nil {
            t.Fatal(err)
        }
        got := b.Category()
        if cat != got {
```

```
            t.Errorf("want category %q, got %q", cat, got)
        }
    }
}
```

(Listing 23.2)

For the invalid input case, we can use some large number that doesn't correspond to any of our defined categories:

```
func TestSetCategoryInvalid(t *testing.T) {
    t.Parallel()
    b := bookstore.Book{
        Title: "For the Love of Go",
    }
    err := b.SetCategory(999)
    if err == nil {
        t.Fatal("want error for invalid category, got nil")
    }
}
```

(Listing 23.2)

In the `bookstore` package, we'll need to add the constants and the map of valid categories:

```
type Category int

const (
    CategoryAutobiography Category = iota
    CategoryLargePrintRomance
    CategoryParticlePhysics
)

var validCategory = map[Category]bool{
    CategoryAutobiography: true,
    CategoryLargePrintRomance: true,
    CategoryParticlePhysics: true,
}
```

(Listing 23.2)

We need to update the `Book` struct type to change the `category` field from a string to a `Category`, and make the corresponding changes to the `Category` and `SetCategory`

methods:

```
func (b *Book) SetCategory(category Category)
    error {
    if !validCategory[category] {
        return fmt.Errorf("unknown category %v", category)
    }
    b.category = category
    return nil
}

func (b Book) Category() Category {
    return b.category
}
```

(Listing 23.2)

And that's it! We now have a nice, easily extensible set of categories and a way to validate them.

Takeaways

- Although type checking is enforced by the compiler on assignments to struct fields, there's no built-in way to *validate* what we assign to them.
- So if we want to protect users from accidentally assigning an invalid value (for example, a negative price), we need to provide some *accessor* method that can check the value before assigning it.
- This would have to be a pointer method, since without a pointer anything it assigned to the struct field wouldn't persist: it would just be modifying its own local copy of the struct that then gets thrown away when the method returns.
- There are two behaviours for validating accessor methods that need to be tested: the "valid input" behaviour, and the "invalid input" behaviour.
- Even if there's a validating accessor method, that doesn't actually prevent users from setting any value they like on an exported struct field, bypassing the method altogether.
- So if we want to ensure that certain fields of the struct are *always valid*, we need to make those fields *unexported*, and only available via accessor methods: in this case, we'll need to provide a method to *get* the value as well as one to set it.
- However, `cmp.Equal` can't look at unexported fields (nor can anyone else outside the package), so we can pass the option `cmpopts.IgnoreUnexported` to tell it not to even try.

- Just as you can create always valid fields by unexporting the field names, you can create *always valid structs* by unexporting the name of the struct itself.
- That implies some kind of *constructor* function, because users can't create literals of the struct (they'd need to use its name, and they're not allowed to).
- It's fine for constructors to return types with unexported names (indeed, it's essential for always valid structs): it's only the *name* users can't refer to, not the *value*.
- When you want to restrict values to one of an allowed set, a map of `bool` is a nice way to do it, because you can write something like `if validCategory[category]`, and the map lookup itself gives you the right answer, with no extra code.
- Another good way to protect users from mistakes is to use the `const` keyword to define some named *constants* that users can refer to: if they misspell the name, or refer to one that doesn't exist, the compiler will catch this.
- A good example is the set of HTTP status constants defined in `net/http`: it's not inherently obvious without context what the integer literal `200` means, and you could get it wrong, but `http.StatusOK` is unequivocal, and the compiler will tell you if you mistyped it.
- Often it doesn't matter what the values of the constants *are*, only that they're distinct, and in that case we can use integer constants and `iota` to auto-assign values to them.

11: Opening statements

Great work on your project at Happy Fun Books! You have some experience now of writing useful Go programs, so let's take a little time to explore the Go language itself in more detail. This will help you gain a deeper understanding of the code you've already written, and how it works.

In the next few chapters, we'll explore some features of Go syntax that let you define the *control flow* of your programs: that is to say, how they are *executed*. Some of these features will already be familiar, but you'll learn more about them. Others will be new, and you might like to try incorporating them into your existing programs.

So what is control flow? You can imagine a very simple program where "control" just "flows" from the first line of the program, in sequence, to the last. Some programs work exactly this way, but most have a more complicated control flow. Maybe instead of a simple sequence, they *repeat* a sequence multiple times, or perhaps execute different sequences depending on some condition.

Before we get on to more complicated types of control flow in Go, let's start with the basics.

Statements

The most basic element of control flow in a Go program is the *statement*. A statement is simply an instruction to Go to do something. Here's an example:

```
b = Book{Title: "The Making of a Butterfly"}
```

Thanks to your sterling work on the bookstore project, we have available a struct type called `Book`, to represent a book, and it has a string field called `Title` that stores the title of the book.

Declarations

The example above is an *assignment* statement: it assigns a literal value of the `Book` struct to a variable named `b`, which must already exist. How does a variable like `b` come into existence? By means of a *declaration*. For example:

```
var b Book
```

This introduces a new variable `b` that can hold `Book` values. A declaration doesn't actually make anything happen when your program runs: instead, it asks the Go compiler "please reserve some space for a variable of a certain type that I'm going to do something with later".

A declaration isn't technically a statement, because it doesn't produce any effect at run time. It's worth knowing that in Go, almost all our code must be inside some function, and no statements are allowed outside functions (at *package level*). The only kind of code allowed outside a function is a declaration, and that can be useful, as we'll see later on in this book.

What is assignment?

Assignment statements *assign* some value to a variable, as we've seen, but what does that mean in practice? You could think of it as giving a *name* to the value. For example, the `Book` value in our earlier example can, following the assignment statement, be referred to using the name `b`. Any time we use the name `b` in an expression from now on, it will be "translated" into the literal `Book` value that we assigned to `b`.

Another way of thinking about variables is as little boxes, that contain values of a certain type. Assignment is the act of putting a value into the box. If there was anything in the box beforehand, it gets replaced by the value we assign. What happens if you declare a variable and then assign two different values to it in succession?

```
var name string
name = "Kim"
name = "Jo"
```

This is perfectly all right, and following these two assignments, the `name` variable has the value `"Jo"`; this second value has "overwritten" the first. You can assign to a variable as many times as you like, and it will always have the most recent value you assigned to it.

Short variable declarations

It's so common in Go to want to declare a variable and then assign a value to it that there's actually a special syntax form to do just this: the *short variable declaration*. It looks like this:

```
b := Book{Title: "The Making of a Butterfly"}
```

Note the `:=` instead of plain `=`. This both *declares* a new variable `b`, and *assigns* it a `Book` literal, in a single statement. You can think of the colon (`:`) as meaning "Here comes a new variable: please act as though I already declared it."

Since we haven't explicitly declared its type, how does Go know what type the `b` variable should be? The answer is, the same type as the value we're assigning to it! The compiler can *infer* the type `Book` from the value on the right-hand side of the assignment.

GOAL: Write a Go program that uses a short variable declaration to declare and initialise a variable `eggs` to the value 42. Print the value of this variable, then assign the new value 50 to `eggs` and print the variable again to show that its value has changed.

Assigning more than one value

When you have several assignments to do, Go has a nice feature to help you: the *tuple assignment*. A *tuple* is an ordered list of things (as in "quintuple", "sextuple", and so on).

Suppose we have three variables `a`, `b`, and `c`, and we want to assign them the values `1`, `2`, and `3` respectively. Of course we could write three assignment statements, but we're all busy people. It's good to be able to do this job in one tuple assignment:

```
a, b, c := 1, 2, 3
```

It's best to reserve tuple assignments for assigning groups of *related* values: don't just use it as a way of avoiding separate assignment statements for unrelated variables. By using a tuple assignment, you're telling the reader "These are all the same kind of thing, so I won't test your patience by listing them separately: I'm assigning them all in one go".

Similarly, it's a good idea to keep your assignment statements short and clear. If your tuple assignments are long and complicated, it makes your code hard to read. If it isn't completely clear at a glance which variable is getting what value, then split up your assignment into multiple statements for clarity.

Perhaps the most common use for multiple assignments is when a function returns multiple values (and we'll talk more about functions in a later chapter). For example, suppose you have a function `getPosition` that returns your current position on Earth as a pair of co-ordinates: latitude and longitude, let's say. We can assign these two values to two variables using a tuple assignment statement:

```
lat, long := getPosition()
```

If your distance from the centre of the Earth matters too, you'll need three co-ordinates:

```
x, y, z := getPosition()
```

There are other things that can give us multiple values in Go. One example that you may recall from earlier chapters is a map index expression:

```
price, ok := menu["eggs"]
// ok is true if "eggs" was in the map
```

The blank identifier

Do we actually even need a *variable* in an assignment statement? The answer is no; there's a special identifier called the *blank identifier*, spelled `_` ("underscore"), and we can assign things to it.

Why would we want to use the blank identifier? It doesn't seem particularly useful, because even though we can put things "into" it, we can't get them out again:

```
_ = "Hello"
fmt.Println(_)
// compile error: cannot use _ as value
```

Well, sometimes we're not interested in one or more of the multiple values that Go gives us. For example, if we just want to see whether a given key is in a map, we really only want the `ok` value, and we won't be actually using the price of eggs at all. The problem is that Go won't let you declare a variable if you never reference it again:

```
price, ok := menu["eggs"]
// compile error: price declared but not used
```

You might think that maybe we can leave `price` out and just put `ok` on the left-hand side of the assignment:

```
ok := menu["eggs"]
```

But the problem with *this* is that Go assigns values to variables in left-to-right order: the first variable will receive the first value from the map index expression, and in this case, that will be the price of eggs, and not the `bool` value we wanted.

So we need some way to write a *placeholder* where a variable would go, but we're explicitly telling Go "Hey, I'm not interested in the value that corresponds to this variable: just throw it away". That's what the blank identifier does:

```
_, ok := menu["eggs"]
```

You can use the blank identifier more than once in the same statement. For example, if you're only interested in your `z`-position in space:

```
_, _, z := getPosition()
```

Increment and decrement statements

Another kind of statement in Go is the *increment* statement, which increases the value of some numeric variable by 1:

```
x++
```

Note that this is a statement, not an expression, so it doesn't have a value. We can't assign the "result" of the increment operation to something:

```
x := y++
// syntax error: unexpected ++ at end of statement
```

Conversely, *decrement* statements decrease the value of the named variable by 1:

```
x--
```

`if` statements

We can do a lot with simple statements in Go, but for our programs to be really useful, we need to be able to make different decisions in response to circumstances: that is to say, we need *conditional* statements.

For example, we might want to do something only if a certain condition holds. In Go, we would use an *if statement* for this.

An `if` statement begins with the keyword `if`, followed by some expression whose value is true or false. This is the *conditional expression* which determines whether or not the code in the `if` statement will be executed. Here's an example:

```
if x > 0 {
    fmt.Println("x is positive")
}
```

What happens when this program runs? First of all, the conditional expression `x > 0` is evaluated. Its value is `true` if the variable `x`'s value is greater than zero, or `false` otherwise.

If the condition is false, then nothing happens, and the program proceeds to the next statement, whatever it happens to be. But if it's true, then the code *block* inside the curly braces is executed, and the program prints out the message `"x is positive"`.

Note that the indentation here helps us visualise what's going on. Normally, execution proceeds through a function in a straight line "downwards", if you like, following the left margin of the code. But the code inside the `if` block is indented, showing that the path of execution changes direction if the condition is true.

The happy path

This gives us a clue to a helpful way to write `if` statements. Often, the control flow of a function has a "normal" path, the path it takes if there are no unusual or exceptional circumstances (errors, for example). This is sometimes called the *happy path*: it's what happens if everything goes right.

Consider a function which validates some input; perhaps it needs to ensure that a supplied number is positive. You already know how to write a suitable `if` statement for this condition. So here's one possible way to write the function:

```
func valid(x int) bool {
    if x > 0 {
        fmt.Println("Yay! That's a valid input.")
        return true
    }
    fmt.Println("Nope, x is zero or negative")
    return false
}
```

This is perfectly fine and will do what we need it to. But is there a better way to *write* this function, without changing its actual behaviour? This kind of code improvement, by the way, is called *refactoring*. Can we refactor this function to read more naturally?

To answer that, let's see what happens if we add another validation condition. As well as requiring that `x` be positive, we will require that it also be even (that is, divisible by 2). Go has a handy arithmetic operator we can use for this, called the *remainder operator* (`%`). It gives the remainder when one number is divided by another, so `x%2` gives the remainder when `x` is divided by 2. If `x` is even, that should be zero, so the conditional expression we need is:

```
x%2 == 0
```

Where to write it? We could add another `if` statement inside the block which is executed on condition that `x` is positive:

```
if x > 0 {
    if x%2 == 0 {
```

```
        fmt.Println("x is positive and even.")
        return true
    }
    fmt.Println("Positive, but odd. Too bad.")
    return false
}
```

But you can see that we now have two levels of indentation, and the code that executes if x is positive and even is now way over to the right-hand side. Add a couple more conditions, and this would become quite hard to read, especially on narrow screens or pages. Also, there's no real clue to the reader as to which way control is expected to flow: no visual "happy path".

Let's refactor this by simply *flipping* (inverting the sense of) the conditional expressions:

```
if x <= 0 {
    fmt.Println("Nope, x is zero or negative")
    return false
}
if x%2 != 0 {
    fmt.Println("Positive, but odd. Too bad.")
    return false
}
fmt.Println("Yay! That's a valid input.")
return true
```

The logic of this function is exactly the same as the previous one, but now it's easier to read. The happy path follows a straight line down the left margin. The only excursions away from it happen if there is some problem with `x`, and those are dealt with inside indented blocks, visually separating them from the expected flow of control.

When deciding how to write your `if` statements, this is a helpful rule: *Left-align the happy path*. It minimises indentation and provides a clear visual guide to the reader about how you expect the program to proceed.

GOAL: Write a function `bigger` that takes two `int` parameters and returns whichever is the larger.

`else` branches

So far, we've imagined the flow of control as looking a little bit like a highway down the page, with `if` statements creating possible diversions onto a minor road that branches off the highway, perhaps rejoining it later.

But sometimes the highway itself needs to fork, and take one set of actions if the condition is true, and another set otherwise. This is where the `else` keyword comes in.

```
fmt.Println("Let's see what the sign of x is:")
if x <= 0 {
    fmt.Println("x is zero or negative")
} else {
    fmt.Println("x is positive")
}
fmt.Println("Well, that clears that up!")
```

For example, if x is 12, we'd see this output:

```
Let's see what the sign of x is:
x is positive
Well, that clears that up!
```

The flow of control is again clear and easy to follow. If the condition `x <= 0` is true, then the message `"x is zero or negative"` is printed. Otherwise, the other message is printed. Whereas an `if` statement on its own just does or doesn't do something based on the condition, an `if ... else` statement does one thing *or* another (but never both).

Early return

In fact, `else` isn't much used in Go. Why not? Well, it often simply isn't needed, because in the event of some `if` statement diverting it from the happy path, a function returns immediately:

```
if x <= 0 {
    fmt.Println("Nope, x is zero or negative")
    return false
}
```

It's clear that if the function gets past this point, the condition *must* have been false, or we would have returned already. So `else` isn't needed when an `if` block ends with a `return` statement. This pattern is known as *early return*, and it can help make your functions more concise and easy to read.

Conditional expressions

Now we know how `if` statements can use conditional expressions to make decisions affecting the control flow of a program, let's take a closer look at those expressions. What kind of expressions are allowed in `if` conditions?

You've seen one example already in this chapter: a *comparison* expression, such as `x <= 0` (which is true if `x` is less than or equal to zero). Similarly, we can use other comparison operators: `==` (equal to), `!=` (not equal to), `>` (greater than), `<` (less than), and their counterparts `<=` (less than or equal to) and `>=` (greater than or equal to).

And, or, and not

We're not limited to simple boolean expressions like `x > 0` and `x%2 == 0`. We can combine them together using *logical* operators: the equivalent of "and", "or", and "not" in English sentences.

For example, to express the condition that `x` be both positive *and* even, we can write:

```
if x > 0 && x%2 == 0 {
    fmt.Println("positive and even")
}
```

The `&&` operator means "and". Similarly, if we want a condition that's true if one expression *or* another is true, we use the `||` operator to signify "or":

```
if x > 0 || x%2 == 0 {
    fmt.Println("positive or even (or both)")
}
```

Note that in English, "or" is ambiguous: it can sometimes mean "exclusive or" (one possibility or the other, but not both), and sometimes "inclusive or" (one or both of the possibilities), depending on how it's used.

"You can learn Go or Python" is certainly not exclusive: you can learn both (and you should). But "you can get up early and meditate, or you can stay in bed until noon" is exclusive: you can't take both options, unless you've mastered astral travel.

In Go, though, `||` is unambiguous: it's an inclusive "or", so the expression `a || b` is true if *any* of the following conditions is true:

- `a` is true
- `b` is true
- `a` and `b` are both true

The only case in which it can be false is when `a` and `b` are both false.

For conditions which are true if some expression is *not* true, we use the `!` (not) operator:

```
if !ok {
    fmt.Println("That is not okay")
}
```

`bool` variables

As you may recall, values which can be either true or false are called *boolean* values, and the Go data type which represents them is named `bool`.

We can use the predeclared constants `true` or `false` anywhere a boolean value is needed, including in `if` conditions. However, there's no point in writing `if true ...`, because its block will always be executed, so we may as well leave out the `if true` part. Similarly, an `if false ...` statement will never be executed, so we can leave it out altogether.

However, a `bool` variable on its own is a common conditional expression:

```
_, ok := menu["eggs"]
if ok {
    fmt.Println("Eggs are on the menu!")
}
```

GOAL: Write a function `xor` ("exclusive or") that takes two `bool` parameters and returns `true` if exactly one of the parameters is true, or `false` otherwise. (Just to be clear, if both parameters are true, the function should return `false`.)

Compound `if` statements

It's very common in Go to assign or compute the value of some variable, and then use an `if` statement to take some decision about it. So common, indeed, that Go provides a special form to make this easier. We can write the assignment statement *after* the `if` keyword, but before the condition, separated from it by a semicolon (`;`):

```
if _, ok := menu["eggs"]; ok {
    fmt.Println("Eggs are on the menu!")
}
```

This is a little confusing to read at first, but it's really exactly the same as the previous example. We first of all assign the variable `ok` depending on whether the map lookup succeeded or not, and then we conditionally execute some code if the variable is true. Here's the same idea expressed in symbolic form:

```
if STATEMENT; CONDITION {
    ...
}
```

By using this form, you're telling the reader that the `ok` variable is only needed within this one `if` block, and you won't be using it afterwards. In fact, the rules of *scope* for Go variables mean that, in this form, the `ok` variable *can't* be used outside the `if` block. You can think of it as being local to the block. Trying to access it afterwards gives a compiler error:

```
    if _, ok := menu["eggs"]; ok {
        fmt.Println("Eggs are on the menu!")
    }
    fmt.Println(ok)
    // error: undefined: ok
```

You'll often see code like the following, that calls some function and then takes a decision based on its result, using a compound `if` statement:

```
if err := doStuff(); err != nil {
    fmt.Println("oh no")
}
```

This is more concise than writing the assignment as a simple statement, and then following it with an `if` statement, and it restricts the scope of `err` to just this block. If an `err` variable already exists in the enclosing scope, it will be *shadowed*: that is, inside the `if` block there's a new and distinct variable `err` that just happens to have the same name, but its value will be whatever is returned by `doStuff()`. After the block, though, the "outer" `err` variable will still have its original value.

If you think this sounds confusing, you're right. It's best to avoid shadowing variables like this. *Go* won't be confused, because it has unambiguous rules about scope, but *we* might, because it's not obvious just from looking at this code what is going on. It's easy to accidentally write a bug by thinking you've updated the value of some variable in the outer scope when you've shadowed it instead.

Similarly, you should avoid making a compound `if` statement do too much. A straightforward function call and assignment is fine, but if that function call is complex and lengthy, then by the time we get to the end of it, we may have forgotten how it started. This is an example of the kind of thing that doesn't help readability:

```
if err := apply(func(x int) error {
    if x <= 0 {
        return errors.New("prognosis negative")
    }
    return nil
}, -1); err != nil {
    fmt.Println("some error happened")
}
```

By the time we get to reading the line `}, -1); err != nil {`, we're getting confused. A good rule to follow here is that if your compound `if` statement is short and fits on a single line, it's okay, but if it's longer than that, or just not clear, break it up into multiple statements.

Takeaways

- Go programs consist essentially of a list of *statements* of various kinds, which are instructions to the computer to do something.
- An *assignment* statement associates a value with a variable.
- A *declaration* statement introduces a variable that we will use later (perhaps by assigning to it, for example).
- There's a special kind of statement that combines declaration and assignment, called a *short variable declaration*: for example, `eggs := 42`.
- We can make multiple assignments in a single statement using what's called a *tuple assignment*, like this: `a, b, c := 1, 2, 3`.
- When you're not interested in one of the values in a tuple assignment, you can ignore it using the *blank identifier*, `_`, like this: `_, ok := menu["eggs"]`.
- *Increment* and *decrement* statements adjust the value of some numeric variable up or down by 1: for example, `x++` or `y--`.
- An `if` statement controls whether a certain set of statements will be executed or not, depending on the value of some *condition*.
- In Go, we tend to structure our programs to *left-align the happy path*: that is, `if` blocks tend to deal with errors or exceptional circumstances, leaving the straight-line control flow as the expected path if everything goes according to plan.
- In principle, we can also execute some block of statements when the `if` condition is *not* true, using the `else` keyword, but this isn't much used in practice; we prefer to have the `if` block return instead.
- A *conditional expression*, such as the condition for an `if` statement, is an expression that evaluates to a *boolean* value (`true` or `false`).
- Conditional expressions can be *comparisons*, using == or `!=`, and other operators such as `<`, `>`, `<=`, and `>=`.
- We can also combine expressions using the *logical* operators `&&` (and), `||` (inclusive or), and `!` (not).
- The simplest kind of conditional expression is just a `bool` variable on its own, or even the literal values `true` or `false`.
- There's a special form called a *compound `if`* that combines an assignment and a condition controlling the execution of a block, but it's never necessary to use this, and it can lead to confusing code.

12: Switch which?

We saw in the previous chapter that our Go program can take either *one* optional path, using `if`, or *two* paths, using `if` and `else` together. But what if there are *more* than two options?

The `switch` statement

Suppose you're interested to know whether a certain number is positive, negative, or zero. You could perfectly well write a series of `if` statements to find this out, and sometimes that's fine. But this becomes verbose and complicated as the number of choices, and possible code paths, grows.

What we want is a conditional statement that lets us specify any number of *cases* (that is, possible conditions), linking each case to a different code path. Go's `switch` statement does exactly this. Here's an example:

```
switch {
case x < 0:
    fmt.Println("negative")
case x > 0:
    fmt.Println("positive")
default:
    fmt.Println("zero")
}
```

We begin with the keyword `switch`, followed by an opening curly brace, and then a list of *cases*, and a final closing curly brace.

Switch cases

Each case is introduced by the keyword `case`, followed by some conditional expression (for example, `x < 0`) and a colon.

We can then supply some statements to be executed if the case condition is true (for example, print out a message).

When Go encounters a `switch` statement like this, it looks at the first case and evaluates the conditional expression there. If this expression is true, the statements for this case are executed, and that's the end of the `switch` statement; control moves on to the next part of the program.

If the expression is false, execution moves on to the next case, and so on, until it finds a case that matches (that is, whose conditional expression evaluates to `true`). If no case matches, then nothing happens and the program moves on.

It's important to know that, once a case has been matched, Go doesn't examine any further cases, and skips to the end of the statement instead. You might think that *all* matching cases are executed, but that's not how it works: only the *first* matching case is executed.

There's a special statement to change this behaviour: writing `fallthrough` at the end of a case tells Go to execute the statements in the next case, instead of skipping to the end. But `fallthrough` is rarely used, or needed; still, now you know about it.

The default case

There is a special case named `default`, which matches only if no other case does. In the previous example, if `x` is neither less than zero (the first case), nor greater than zero (the second case), then the only possibility left is that `x` is zero, which is the `default` case.

You should always supply a `default` case if you can: this makes your programs clearer and more reliable.

`switch` is a very convenient way to write program logic that needs to take one of N possible paths depending on a set of conditions, and it's more concise than writing the corresponding `if ... else if ... else if ...` statements. But note that your `switch` cases should all be related. Don't just use `switch` as a way of avoiding writing multiple unrelated `if` statements; that leads to confusing code.

Switch expressions

It's very common to want to take different actions depending on the value of some variable or expression. For example, suppose you wanted to determine if `x` has the

value 1, 2, or 3. A concise way to do this is to use a *switch expression*, which goes after the `switch` keyword but before the curly brace introducing the list of cases:

```
switch x {
case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two")
case 3:
    fmt.Println("three")
}
```

We say this statement switches *on* `x`, that is, the different cases represent different possible values of the switch expression `x`. You can use any Go expression here instead of `x`, providing that it evaluates to a value of the same type as your switch cases (in our example, that's implicitly `int`, because the constant literals `1`, `2`, and `3` are integers).

You can supply multiple values in the same case, as a comma-separated list. If any of the values match, the case will be triggered:

```
switch x {
case 1, 2, 3:
    fmt.Println("one, two, or three")
...
}
```

GOAL: Write a function `greet` that takes a string representing a person's name and returns a different string depending on the name. If the name is `Alice`, return `"Hey, Alice."` If the name is `Bob`, return `"What's up, Bob?"`. For any other name, return `"Hello, stranger."`

Exiting a switch case early with `break`

It sometimes happens that, inside the code for a switch case, we want to leave early, without executing the rest of the case. The `break` keyword will terminate the current case and jump to the code immediately following the `switch` statement:

```
switch x {
case 1:
    if SomethingWentWrong() {
        break
    }
    ... // otherwise carry on
}
```

You can think of this as the case "breaking out" of the `switch` statement, like a prisoner unwilling to serve the rest of their sentence, perhaps.

Loops

You're learning lots of useful techniques to control the flow of execution in your Go programs. Ordinary statements flow from one to the next, while `if` and `switch` statements take different paths depending on the value of some variable or expression.

But what about if we want to *repeat* certain statements, or not? For example, suppose we want to print some message 10 times. We could simply write a sequence of ten identical `fmt.Println` statements. But this is both verbose and inflexible.

Instead, we would like to have a *single* print statement, and execute it a varying number of times. Try as you will, you can't create a program like this using just `if` or `switch` statements. We'll need a new keyword.

The `for` keyword

A set of statements which can be repeated is called a *loop*, and in Go loops are introduced by the keyword `for`. You can think of it as meaning "do the following *for as long as* some condition holds".

```
for x < 10 {
    fmt.Println("x is less than 10")
    ... // update X, perhaps
}
```

We begin with the `for` keyword, followed by an (optional) conditional expression, and an opening curly brace. Then follows an (optional) set of statements, and a final closing curly brace.

What happens when Go encounters such a loop in your program? First of all, the conditional expression is evaluated. If it is true, then the statements inside the curly braces are executed.

So far, this is just like an `if` statement, but there's a twist. When Go reaches the end of the loop, it evaluates the condition *again*. If it *still* evaluates to `true`, then the statements in the loop body are executed again, and so on forever, or at least until the condition is false.

Forever loops

Since omitting a condition is equivalent to specifying the condition `true`, which is always true, you can see that a `for` loop without a condition will execute forever. (Or, at

least, until you turn the computer off or otherwise interrupt the program.) We might call this a "forever loop".

Is such a loop useful? Actually, yes! There are many programs whose job it is to repeat some process forever: manufacturing processes, for example. A welding robot on an assembly line just repeatedly performs the same sequence of welds as long as the line is running.

Similarly, network servers such as HTTP webservers run an endless loop that waits for a request to arrive, services the request, and then goes back to waiting. There are, of course, ways to interrupt or shut such a program down, but we don't need to specify them as a terminating condition for the loop.

Using `range` to loop over collections

More commonly, though, we want a loop to execute once for each element in some collection of data. Imagine a payroll program, for example, whose job is to calculate and generate a payslip for each employee of a company.

It wouldn't make sense to use a forever loop for such a program, because it would reach a point at which all employees have been paid for the current month, and there is no more work to do.

Instead, it would be convenient to write a loop that executes once for every element in a slice, or some other kind of collection of data. The `range` keyword will do exactly this.

For example, suppose we had a slice variable named `employees`, which contains data about each employee (such as their salary). We could write a loop using `for` and `range` as follows:

```
for range employees {
    fmt.Println("Found another employee!")
}
```

If there are three elements in the `employees` slice, then this loop will execute three times, printing the following output:

```
Found another employee!
Found another employee!
Found another employee!
```

Receiving index and element values from `range`

The loop in our previous example isn't that useful, however, because although it can execute a set of statements for each employee, it doesn't have any *data* about the employees (such as their salary). So for something like a payroll program, we'll need a way not

only to repeat the loop once for each element, but to give the loop access to the element itself.

As you may recall from our earlier work at Happy Fun Books, `range` can return up to two values from a slice: the *index value* of each element (0, 1, 2, 3...) and the *element value* (the element itself).

For example:

```
for i, e := range employees {
    fmt.Println("Employee number %d: %v", i, e)
}
```

Given suitable data in the `employees` slice, this might print out something like the following:

```
Employee number 0: Zhang Wen
Employee number 1: Yu Luoyang
Employee number 2: Qiu Zhenya
...
```

Often, we're not interested in the index value, so we ignore it using the blank identifier (`_`):

```
for _, e := range employees {
```

Or if we *only* want the index value, we just needn't bother to supply any variable to receive the element value:

```
for i := range employees {
```

This is an extremely useful and common pattern for looping over data. That is, executing a set of statements once for each element in a slice, or each key-value pair in a map, or each rune in a string.

GOAL: Write a function `total` that takes a slice of `int` and returns the sum of all the slice elements. For example, if the input is `[]int{1, 2, 3}`, the function should return 6.

Conditional `for` statements

Let's go back to the *conditional* `for` loop that we introduced earlier in this chapter, and that keeps executing as long as some specified condition is true. If we omit the condition altogether, as we said, that is equivalent to specifying the condition `true`, and the loop executes forever. But we can write more interesting conditions.

For example, suppose we want to print out the numbers 0 through 9. We can use some integer variable `x`, and use a `for` statement to repeatedly print the value of `x` and then increment `x`:

```
x := 0
for {
    fmt.Println(x)
    x++
}
```

This behaves exactly as you'd expect, printing the values 0, 1, 2, 3, and so on in sequence. But the sequence never terminates, because we never told it to! Since we're only interested in numbers less than 10, we can add a condition to the `for` statement which specifies that the loop should only keep executing as long as `x` is less than 10. That's straightforward:

```
for x < 10 {
```

Now the loop does exactly what we wanted, which is to print out the numbers 0 through 9 and then stop.

Init and post statements

This kind of loop is so common, in fact, that Go provides us with a special form to write it conveniently. We usually want to *initialise* some variable to its starting value, then execute some code as long as its value meets some condition, and then *update* the value in some way (for example, incrementing it).

So we can add an initialising statement before the condition (known as an *init statement*), and an updating statement, that follows the condition, and executes *after* the loop body (known as a *post* statement for this reason). The init statement, condition, and post statement are separated by semicolons. Here's our previous example expressed in this form:

```
for x := 0; x < 10; x++ {
    fmt.Println(x)
}
```

This code produces exactly the same output as before, but it's more compact. Notice that the initialising statement (`x := 0` has moved inside the `for` statement, before the condition, and the updating statement (`x++`) has also moved inside the `for` statement, after the condition.

You can see now that the earlier loop example (`for x < 10`) was actually just a special case of this three-part `for` statement form, where the init and post statements were omitted, leaving only the conditional expression.

This might all seem a little over-complicated to you, but don't worry. Three-part `for` statements like this are relatively uncommon in Go, because what we usually want to do is either loop over some data (in which case we can use `range`) or just loop forever, which doesn't require a condition at all.

GOAL: Write a function `evens` that prints out all the even numbers between 0 and 100 inclusive, using a three-part `for` statement.

Jumping to the next element with `continue`

We've seen how to use `for` loops to repeat a linear sequence of statements, but what if it's not that simple? Maybe we only want to perform some operation on certain elements of the data, and not others, for example.

Our payroll program might want to generate paychecks only for current employees, for example, and not those who have resigned or retired. We could write something like this:

```
for _, e := range employees {
    if e.IsCurrent {
        e.PrintCheck()
    }
}
```

That's okay, but not ideal; we have an extra level of indentation, inside the `if e.IsCurrent` block, and, as you'll recall from earlier in this book when we talked about the "happy path", we would like to keep indentation to a minimum to make our code easy to read.

What we would like is a way, in the case of non-current employees, to say "Just do nothing in this case, and continue the loop with the next element instead". That's exactly what the `continue` keyword does:

```
for _, e := range employees {
    if !e.IsCurrent {
        continue
    }
    e.PrintCheck()
}
```

Now the happy path runs in a straight line down the left margin, and the only indented code is for handling exceptional or invalid situations, which is just what we wanted. You can imagine that there might be several different reasons to skip employees, each of which could trigger a `continue`.

GOAL: Write a function `nonNegative` that takes a slice of `int` and prints only the non-negative elements. For example, if the input is `[]int{2, -1, 0, -4}`, it should print only 2 and 0.

Exiting loops with `break`

You already know that you can control when a loop ends, by adding a conditional expression (or a `range` expression) to the `for` statement. But sometimes we need to terminate a loop in some situation that we can't detect in advance; something that happens during the loop, for example.

We've seen that the `continue` keyword will skip the rest of the *current* loop and go on to the next, but that's not quite what we want here. We want to jump out of the loop altogether, as though we'd reached the terminating condition and the loop's work was done.

Often it's enough to simply `return` from the current function; `return` statements can go anywhere in a function, including inside loops. Since `return` returns immediately, this would have the effect of bailing out of the loop.

But when the function needs to do further work after the loop is complete, we can't use `return` to exit the loop; it would exit the function too. In the chapter earlier in this book on `switch` statements, we came across the useful keyword `break`, for exiting the statement early, and we can use it to exit loops too.

For example, suppose we want to stop printing checks if the company has run out of money during the loop (this occasionally happens, but let's hope not too often). We could write something like this:

```
for _, e := range employees {
    if MoneyLeft() <= 0 {
        fmt.Println("Oops, out of cash!")
        break
    }
    ... // otherwise, print check
}
```

As soon as Go encounters a `break` statement, it immediately stops the loop and jumps to the code immediately following the closing brace at the end of the `for` statement.

Controlling nested loops with labels

As you know, a loop in Go can contain any code, including other loops. It might occur to you to wonder what happens if you use the `break` or `continue` keywords inside a loop that's already inside another loop (this is known as a *nested loop*).

Well, `break` on its own just breaks out of the *current* loop; that is to say, the loop that we're inside. It has no effect on any enclosing loop. Similarly, `continue` continues the current loop. So what do we do if we want to break out of *both* loops from inside the *inner* loop?

The answer is that we can use a *label*. A label in Go is simply a way of giving a name to a particular location in the code, so that we can refer to it later. The syntax for a label is just the name of the label followed by a colon. For example:

```
outer:
```

If we have nested loops we need to break out of, we can do it by placing a label just before the start of the outer loop, and then using `break` with that label in the inner loop:

```
outer:
    for x := 0; x < 10; x++ {
        for y := 0; y < 10; y++ {
            fmt.Println(x, y)
            if y == 5 {
                break outer
            }
        }
    }
```

If we wanted to `continue` the outer loop instead of exiting it, we could have written `continue outer`.

This is a slightly contrived example, and in fact it's very uncommon to see labels used in Go programs at all. For one thing, nested loops are already rather confusing and hard to read, so we tend to avoid them where possible.

Also, any kind of "jumping" control flow like this makes the program more difficult to understand. If you find yourself needing to use labels in your Go code, it might be a hint that you could clarify your program by refactoring it to do things a different way, that doesn't need labels.

As if jumping around with `break` and `continue` weren't bad enough, there's another Go keyword `goto` that will simply jump directly to the specified label. As you can imagine, this can make things *really* hard to follow. You're unlikely to have a good reason to use `goto` in Go, and you should avoid it if you want to write clear, readable code.

Takeaways

- A `switch` statement is a concise way of selecting one of several possible *cases*, depending on some condition, without having to write lots of consecutive `if ... else ...` statements.
- Each case in a `switch` statement can have a condition, in which case Go evaluates each case in order and executes the first one for which the condition is true, if any.

- If there is a case called `default`, it is executed if and only if no other case has been matched.
- Instead of conditional cases, we can provide an expression after the `switch` keyword; in this case, Go will evaluate the expression and execute the first case that specifies a matching result.
- To exit a `switch` statement early, before reaching the end of the current block, use the `break` statement.
- To repeat a set of statements a number of times, perhaps forever, or at least until some condition is false, use a `for` statement.
- A `for` loop with no condition simply repeats forever: `for { ... }`.
- To loop once for each element of a slice or a map, use the `range` operator: `for range employees { ... }`.
- To receive the index number and the value for each element, use `for i, e := range employees { ... }`.
- On the other hand, if we just want the element value but not the index, we can write `for _, e := range employees { ... }`.
- And if we only want the index, we receive only that: `for i := range employees { ... }`.
- When we want the loop to repeat as long as some condition holds, we can write, for example, `for x < 10 { ... }`.
- And to repeat a specified number of times, we can manage the loop variable entirely in the `for` statement, like this: `for x := 0; x < 10; x++ { ... }`.
- To skip the current loop index or element and go on to the next, use the `continue` statement.
- To jump out of the loop altogether, use the `break` statement.
- If our program is so complex that it needs *nested* loops, we can use `break` or `continue` with a *label* to jump directly to the labelled statement.

13: Fun with functions

Some programs just execute a single series of statements from start to finish, perhaps repeating some of them in a loop. That's fine, but it's often very useful to be able to organise code into *functions*: named chunks of code that have well-defined inputs and outputs.

Then, if we need to perform the same operation at multiple different places in the program, we don't need to repeat all that code: we can turn it into a function, and *call* the function wherever we need to.

You've already written several functions, so this isn't entirely news to you, but let's take a closer look now at functions and what they can do.

Declaring functions

Before we can call a function we need to *declare* it, which involves telling Go the name of the function, any input *parameters* that it takes, and any *results* that it returns, as well as supplying a *body* for the function which actually does something. We'll see how to do all that in this section.

Firstly, where can we declare functions? The answer is *at package level*, that is to say, within a package (all Go code must be within some package) but outside any function.

A function declaration is introduced by the keyword `func` (meaning "here comes a function!"). Here's a complete example of a function declaration:

```
func double(x float64) float64 {
    return x * 2
}
```

Following `func` is the name of the function (`double`), and a list of its parameters in parentheses.

Parameter lists

Some functions don't need to take any parameters, and that's okay. In that case we would just write a pair of empty parentheses for the parameter list.

If there are parameters, though, we need to give the *type* we expect for each parameter. In our `double` example, there's just one parameter, of type `float64`. We can give the parameter a *name*, too (`x` in our example), so that we can refer to it in the function. (Parameters don't *have* to have names, but if we're not going to refer to them, why would we need to take them in the first place?)

If there's more than one parameter, we write a comma-separated list within the parentheses. For example:

```
func add(x float64, y float64, z float64) float64 {
```

When a sequence of parameters have the same type, though, we can write this in a shorter way:

```
func add(x, y, z float64) float64 {
```

Like the previous function, this takes three `float64` parameters named `x`, `y`, and `z`; we just wrote its signature more concisely by omitting the repeated type name `float64`.

Result lists

Having given Go the name of the function we're declaring, and its list of parameters, we then proceed to say what *results* it returns (if any). Again, we give a comma-separated list of types within parentheses:

```
func location() (float64, float64, error) {
```

Just as with input parameters, we can give these result parameters names, and we'll see why we might want to do that in the next chapter. But first, let's complete our look at function declarations.

The function body

So far, we've told Go what parameters our function takes and what results it returns. This characteristic combination of types is called the function's *signature*. But for the

function to actually do anything, we need to add a *body* to its declaration.

The function body is a list of statements contained within curly braces. For example:

```
func hello() {
    fmt.Println("Hi, I'm the function body!")
}
```

This function takes no parameters and returns no results, but it nonetheless has a body, consisting of the `fmt.Println` statement. A function with an empty body is perfectly legal in Go, but probably not very useful.

Calling a function

It's worth pointing out that merely *declaring* a function doesn't actually cause anything to happen when your program runs. The statements in a function body are not executed until the function is *called*. So how does that work?

We've seen various kinds of control flow features so far in this book, including simple statements, conditional statements, loops, and so on. A *function call* also affects control flow. It causes execution to jump from the place where the function is called, to the start of the function body.

When the function exits, control returns to the statement immediately following the original function call. Calling a function is like saying "Go over there and do this, then come back here for further instructions."

We can call a function by giving its name, followed by some *arguments*: a list of values in parentheses that matches the function's declared parameters. (If we're not supplying any arguments, we simply write a pair of empty parentheses.) For example:

```
hello()
```

This causes the program to jump immediately to the beginning of the `hello` function, execute its body, and then come back here.

Here's another example, this time calling a familiar standard library function and passing some values to it as arguments:

```
fmt.Println(1, 2, 3)
```

Using result values

As we've seen, some functions don't return anything. If they do, though, we usually want to do something with that result. For example, we might assign it to a variable:

```
answer := double(2.5)
```

As you know, this is an *assignment statement* (specifically, a short variable declaration). As with all assignments, we have some variables on the left-hand side (just one in this case) and some values on the right-hand side (at least as many as we have variables to receive them).

In order to execute this statement, Go first of all calls the `double` function with the value `2.5` as its argument. Once the function has finished and returned some value (likely 5), this value is what will be assigned to the variable `answer`.

A function call that returns a value is an *expression*, and like any expression, it can be combined with other expressions:

```
answer := 3 * (double(2.5) + 7)
```

This example works because `double` returns a *single* result value, but if it returned more than one, we couldn't use the call to `double` this way. The only thing we can do with multiple results is to use them in a *tuple assignment*, which you may also recall from earlier chapters:

```
lat, long, err := location()
```

Return statements

So how does a function exit and return control to its caller? If a function is declared with no result parameters, it implicitly exits when the end of the function body is reached. But if we want to cause the function to return before this point (perhaps because some error occurred, for example), we can use a *return statement*.

```
if err != nil {
    return
}
```

This causes the function to return immediately.

If the function is declared to return some results, then we can't just use `return` on its own; it must be followed by the same number of values as the function's signature says it returns.

For example, in a function that returns `(int, error)` we must return an `int` and an `error` value (the compiler doesn't mind what specific values we return, but we must supply something):

```
return 0, errors.New("some error happened")
```

GOAL: Write a function `withdraw` that takes two `int` parameters, representing a bank balance and a withdrawal amount, and returns two results, an `int` representing the balance after withdrawing the requested amount, and an `error` value which indicates whether the withdrawal would leave the user overdrawn.

For example, if `withdraw` is called with a balance of 100 and a withdrawal of 20, it should return a new balance of 80 and a `nil` error value. On the other hand, if `withdraw` is called with a balance of 20 and a withdrawal of 100, it should return a zero balance and a non-nil error indicating that the withdrawal is not allowed.

In other words, your solution should pass these two tests:

```
func TestWithdrawValid(t *testing.T) {
    t.Parallel()
    balance := 100
    amount := 20
    want := 80
    got, err := withdraw.Withdraw(balance, amount)
    if err != nil {
        t.Fatal(err)
    }
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}

func TestWithdrawInvalid(t *testing.T) {
    t.Parallel()
    balance := 20
    amount := 100
    _, err := withdraw.Withdraw(balance, amount)
    if err == nil {
        t.Fatal("want error for invalid withdrawal, got nil")
    }
}
```

(Listing withdraw)

If you get stuck, have a look at my suggested solution.

Function values

Now we know how to declare named functions, and call them. Is that all we can do with functions? Not quite. In Go, *functions are values*. What does that mean? It means we can do anything with a function that we could do with a value. We can:

- assign a function to a variable or struct field
- pass a function as an argument to another function

- return a function as the result from another function

This is very powerful. Let's take an example. Why might we want to assign a function to a variable? Let's refer back to our calculator project example from the beginning of this book. We had functions like `Add`, `Subtract`, and `Multiply`, and we wanted to test them with different inputs.

In fact, we could have tested all three of them with a single test, because all three of these functions have the same signature: they take two `float64` parameters and return a single `float64` result. All we need is a way to specify, for a given test case, which function we want to call.

Here's the kind of test case literal we would like to write:

```
{
    a: 2, b: 2,
    function: calculator.Add,
    want: 4,
}
```

So we need to declare a `function` field on the test case struct, but what's its type? "Function", yes, but that's not enough. We need to specify the exact function signature. Here's how we write that:

```
type TestCase struct{
    a, b float64
    function func(float64, float64) float64
    want float64
}
```

We already had the `a`, `b`, and `want` fields, so we can ignore those. The interesting part is this:

```
function func(float64, float64) float64
```

We're declaring a field named `function` whose type is `func(float64, float64) float64`. Yes, that's a type! And it means exactly what you think it means: "function that takes two `float64` parameters and returns one `float64` result". So any function with this signature would be an acceptable value for `function` here, and indeed `Add`, `Subtract`, and `Multiply` all match this. So we can write a `TestCase` literal with a value like:

```
    function: calculator.Add,
```

How do we declare a function parameter to a function? In just the same way. Here's an example from the standard library, in the `sort` package:

```
func Slice(x any, less func(i, j int) bool)
```

This useful function takes some data `x` of `any` type (usually a slice of something) and a function which tells it how to sort that slice. This function, named `less`, takes two `int` parameters and returns a `bool` result. It will be called by `sort.Slice` repeatedly to compare any two slice indexes `i` and `j`, and it needs to determine which one should sort before the other; that is, which one is *less*.

Function literals

Here's an interesting thing about functions: they don't always need a *name*. We know that any Go value can be written as a literal, and the same applies to function values. This is called a *function literal*. Let's see how it works.

What does a function literal look like? Well, like any literal, it consists of the type name, followed by something in curly braces. Let's write a literal for our calculator example, of a `func(float64, float64) float64`. Instead of supplying some *named* function as the value of `function`:

```
function: calculator.Add,
```

we can write:

```
function: func(a, b float64) float64 {
    return math.Pow(a, b)
},
```

This function takes `a` and `b` and returns the result of raising `a` to the `b`th power, but that's not important. The point is, we didn't need to first declare some named function `Exponent` and then give its name as the value of this field. We created the function on the fly, so to speak, by writing a function literal, and the value of that literal is the function itself.

Here's another example, using `sort.Slice`:

```
nums := []int{3, 1, 2}
sort.Slice(nums, func(i, j int) bool {
    return nums[i] < nums[j]
})
```

We define a slice of integers `nums`, which are in no particular order. We call `sort.Slice` to sort them, and we pass it the slice to operate on, and the `less` function we described earlier in this chapter.

You now know how to read this function literal, and it's very straightforward: it compares, for any indexes `i` and `j`, the corresponding slice elements, and returns `true` if element `i` is less than element `j`, or `false` otherwise. (Note that it's not comparing the values of `i` and `j` directly, but instead the values of the elements at index position `i` and `j`.)

As `sort.Slice` proceeds, it will call this `less` function repeatedly, until it's established that the slice is completely ordered according to this function. We could have written any function we want, of course, to describe any specific ordering that we're interested in. But this particular function specifies ascending numeric order, which is usually what we need.

GOAL: Write a function `apply` that takes an `int` parameter and a `func(int) int` parameter, and calls the supplied function on the supplied value, returning its result. Call it with the following code:

```
apply(1, func(x int) int {
    return x * 2
})
```

Check that the result is 2, as expected.

Closures

Here's an intriguing question, which may already have occurred to you. Let's look more closely at the body of the anonymous function literal in the previous example using `sort.Slice`:

```
return nums[i] < nums[j]
```

What it says makes sense, but where did `nums` come from? This function takes parameters named `i` and `j`, and we use those, but we didn't take a parameter `nums`. This variable exists in a different function: the one that called `sort.Slice`. So how come our function literal can "see" this variable?

Well, our anonymous function is defined inside the calling function, and it can "see" all variables that are in scope in that function, including `nums`. This is referred to as a *closure*.

If you think about it, a plain old ordinary function defined at package level can also "see" variables declared at package level, can't it? And this is just the same. Any function has access to the variables available in the scope where it's defined, and when this is a function literal, we call it a closure *over* those variables. In this example, our function literal is a closure over the `nums` variable.

This might seem a little obscure at first, but it's essential for things like `sort.Slice`. If the `less` function weren't a closure over the slice to be sorted, how would it access that slice at all? We can't pass it in, because the type of the function parameter to `sort.Slice` is fixed: it just takes two `int`s. Closures allow us to solve this problem in a very elegant way.

It's common to want to call some arbitrary function and say "perform some operation on this bunch of data". Now we know we can specify the operation to be performed by

passing a function literal, and we don't even need to pass the data, because the function literal itself can be a closure over it!

You can think of a closure as a little bubble, which traps all the variables in scope at the point where it's defined. Then if you pass that bubble as an argument to some other function, those variables will still be trapped inside the bubble and available for use when it gets there. Well, that's how I think of it, anyway: invent your own analogy!

Closures on loop variables

There's a slightly subtle aspect of the way closures work in Go that can lead to surprising behaviour. Consider this simple program:

```
for _, v := range []int{1, 2, 3} {
    fmt.Println(v)
}
```

What does this print? It seems like the answer should be:

```
1
2
3
```

and indeed it is; you might like to satisfy yourself that this is the case by trying it out.

But suppose we replace the call to `fmt.Println(v)` with a call to an anonymous function literal:

```
func() {
    fmt.Println(v)
}()
```

This is also extremely straightforward, and produces the same result. Now let's do something a little tricky. Instead of *calling* this function literal, let's *append* it to a slice of such functions:

```
funcs := []func(){}
for _, v := range []int{1, 2, 3} {
    funcs = append(funcs, func() {
        fmt.Println(v)
    })
}
```

Nothing has been printed out yet, because we haven't *called* any of these three functions: we've simply stored them in a slice. Now let's loop over that slice calling the functions. This is very easy:

```
for _, f := range funcs {
    f()
}
```

And here's what it prints:

```
3
3
3
```

Wait, what? Why did this produce a different answer? The only change we made was to put off calling the functions until after the first loop had finished. So what's going on? You might like to see if you can figure out the answer before reading on.

We said that a function literal is a closure over certain variables, but what does that really mean? It means that when the function is called, it will be able to access the value of that variable. But *which* value? The value it had at the point the function literal was defined, or the value it has at the moment the function is called?

We already know the answer, thanks to our program: it's the value of the variable when the function is *called*, not when it's *defined*. In our example, we set the variable `v` to the values 1, 2, and 3, in succession. So after the first loop has finished, `v` has the value 3; that's straightforward.

Now we call our three functions, each of which prints out the value of the variable `v`. And we know that's 3. So this behaviour is entirely correct and not surprising at all... once you know that that's how closures work.

In fact, any time you're updating a variable that's referenced by a closure, you need to be careful that you don't inadvertently get the wrong result when the closure is called. This most commonly happens in loops, but it can happen in other places, too.

Cleaning up resources

It often happens that we do something in a Go function that requires some kind of cleanup before the function exits. For example, when we open a file, we must also close it before exit, or that resource will *leak* (that is, the garbage collector won't be able to purge it from memory, because it thinks it's still in use).

The operating system and the Go runtime need to maintain some data about open files, and this takes up memory space, so if we keep opening files and never closing them, eventually we will run out of memory. The same applies to any resource that takes up memory.

To avoid this, we might write something like the following:

```
f, err := os.Open( "testdata/somefile.txt" )
... // do something with f
f.Close()
```

Fine! The resource is cleaned up. But you know that a function can exit at any point by using `return`, not just at the end. This creates a potential problem, doesn't it? What if we exit before `f.Close()` is called? That would mean `f` is never closed, and it will hang around in memory forever. For simple, one-shot programs like CLI tools, that's maybe not so serious. But for long-running programs like servers, it will mean that the program's resource usage keeps growing, until eventually it crashes.

If we're diligent and disciplined programmers, which no doubt we are, we can just try to *remember* to call `f.Close()` at every place the function can return. But nobody's perfect, and if someone else were to make changes to this function, they might not even be aware that they were supposed to close `f` before exiting. If there are several resources to clean up, this could also lead to a lot of duplicated code.

What we really want is some way to say "When this function is about to exit, no matter where or how, *make sure `f` is closed*".

The `defer` keyword

It turns out that Go's `defer` keyword is exactly what we're looking for. A `defer` statement takes some function call (for example `f.Close()`) and *defers* it.

That is, it doesn't call the function *now*, but it "remembers" it for later. And when the function is about to exit for whatever reason, *that's* when the deferred function call actually happens.

Here's what that might look like with our file example:

```
f, err := os.Open("testdata/somefile.txt")
if err != nil {
    return err
}
defer f.Close()
... // do stuff with f
```

What's happening here? First, we try to obtain `f` by calling `os.Open`. This may fail, in which case we don't need to worry about closing the file, because we couldn't open it! So we return an error in that case.

But now we know that we successfully opened `f`, so we immediately `defer f.Close()`. This doesn't close `f`, but it "remembers" that we need `f.Close()` to be called on exit.

And now we can do anything we want in the rest of the function, safe in the knowledge that whatever happens, `f` will be automatically closed, without us having to worry about it.

Multiple defers

It's not common, but it sometimes happens that we need to defer more than one function call. For example, perhaps we open two files, and we want to defer closing each one separately.

This is fine, and we can use the `defer` keyword as many times as we like in a function, and all the deferred calls will be executed when the function exits. It's worth knowing that the calls are executed in reverse order: that is to say, last deferred, first run. This is sometimes referred to as *stacking defers*.

```
defer cleanup1() // executed last on exit
defer cleanup2() // executed first on exit
```

Named result parameters

Recall from the previous chapter that when we're declaring a function's result list, we can give those results names, or not. (Usually we don't need to.) When and why would we want to give them names, then?

One very important reason is *documentation*. After all, source code is written for humans, not computers. If it's not apparent from a function signature what its results represent, giving them names can help. Take our example from earlier:

```
func location() (float64, float64, error) {
```

This function presumably gets your location, if the name is anything to go by, and returns two `float64` results (and maybe an error). But what *are* those two results? We can make that clearer by giving them names:

```
func location() (latitude float64, longitude float64, error) {
```

Now the reader can see that the first value is the latitude co-ordinate (in some agreed co-ordinate system) and the second the longitude. We might have *guessed* that this was the case, but it's nice to see it written down explicitly.

We don't need to do anything special in the function, just because its results are named; we continue to return them as we would any other results:

```
return 50.5897, -4.6036
```

But there's one handy side-effect; the result parameters are now available inside the function for us to refer to, just as though they were input parameters or local variables. So we could assign to them, for example:

```
latitude = 50.5897
longitude = -4.6036
return latitude, longitude
```

Naked returns considered harmful

The Go specification actually allows us to omit the names from the `return` statement in this case, and this would implicitly return whatever the values of `latitude` and `longitude` happen to be at this point. But even though that's legal syntax, it's not good practice.

It's always clearer to specify the exact values or variables you're returning, and there's no benefit to omitting them. So you should avoid writing these so-called *naked returns*, even though you'll sometimes see them in other people's code.

In particular, you should be aware that just because a function has named result parameters, that *doesn't* mean you must write a naked `return`. You can, and should, make your return values explicit.

Modifying result parameters after exit

So `defer` is useful for cleaning up resources before the function exits. Great. But what if this cleanup operation needs to change the function's results?

For example, suppose we have some function that writes data to a file, and returns an `error` result to indicate whether it succeeded.

Naturally, we want to defer closing the file, to avoid leaking it. But that close operation itself could fail, and that would probably mean we've lost the user's data. Suppose we ran out of disk space just before we wrote the last byte, for example, and on trying to close the file we get some error.

What to do? Our function must return the error, in order to let the user know that we lost their data. But the error doesn't happen until the deferred call to `f.Close()`, and at that point the function's result value is already set. Consider this (partial) code:

```
... // we successfully opened f
defer f.Close()
... // write data to f
... // everything good, so return nil error
return nil
```

If the call to `f.Close()` happens to fail, returning some error, there's nothing we can do about it, because the function is going to return `nil` no matter what happens. It's baked into the `return` statement. Or is it?

Deferring a function literal

Because we can defer any function call, we don't have to defer just `f.Close()`. Instead, we could write some function literal that *calls* `f.Close()`, and defer *that*:

```
defer func() {
    closeErr = f.Close()
    if closeErr != nil {
        fmt.Println("oh no")
    }
}()
```

(Note the empty parentheses at the end, after the function literal's closing curly brace. We don't defer a *function*, remember, but a *function call*, so having defined the anonymous function we want to defer, we then add the parentheses to call it.)

This is a significant improvement: we can catch the error returned by `f.Close()`, and if we can't change the result returned by the function, we can at least bewail the situation by printing `"oh no"`. That's something, at least.

Deferring a closure

But we can do even better! If we named this function's error result parameter (let's call it `err`, to be conventional), then it's available for setting inside the function. We can assign to `err` anywhere in the function, and that will be the value that the function ultimately returns.

How does that help in our deferred function literal? Well, a Go function literal is a *closure*: it can access all the variables in the closing scope—including `err`. Because it's a closure on `err`, it can modify `err` before the function actually returns.

So we can opt to overwrite the value of `err` with whatever `closeErr` happens to be:

```
defer func() {
    closeErr = f.Close()
    if closeErr != nil {
        err = closeErr
    }
}()
```

We achieved the seemingly impossible: modifying the function's result *after* it exited (but before it returned). All it took was a combination of named result parameters, `defer`, and a closure.

Note that it wouldn't have been correct to merely set `err` to whatever `f.Close()` returned:

```
defer func() {
    err = f.Close() // this is wrong
}
```

Why not? Because `err` already has some value, specified by the function's `return` statement. We don't want to erase that and overwrite it with, for example, `nil`, if the close operation succeeds. The only time we want to overwrite `err` is if `f.Close()` fails.

You won't need to do anything quite as tricky as this in most functions, but it's occasionally extremely useful.

Variadic functions

So far we've dealt only with function that take a specified number of parameters (for example, `Add` takes two parameters, `a` and `b`). And that's fine: most Go functions you'll ever write will be of this kind, where we know in advance how many parameters to expect.

But some Go functions can take a variable number of parameters; these are called *variadic functions*. For example, you might have noticed that you can pass any number of arguments to `fmt.Println`:

```
fmt.Println(1, 2, 3)
```

So how do we actually write a variadic function? For example, suppose we wanted to extend our calculator program from earlier in this book, so that instead of just adding *two* numbers, we could add any number of numbers.

Let's call this function `AddMany`. What would it look like? Here's the beginning part of the function declaration:

```
func AddMany(inputs ...float64) float64 {
```

Notice that we only declare a *single* parameter, but its type is something strange. It's `...float64`. What does that mean?

The `...` indicates that there can be zero, one, two, or indeed any number of `float64` parameters to the function. Okay, fine. But how can we actually *refer* to the values of those parameters within the function? They don't have names like `a` and `b`: all we have is the single parameter `inputs`.

Well, inside the function body, `inputs` acts like a slice. For example, we can write a `for ... range` statement that loops over each element of `inputs`:

```
for _, input := range inputs {
    // 'input' is each successive parameter
}
```

See if you can do the rest. Use this idea to implement `AddMany` (with a suitable test, of course). Then extend your other functions `Subtract`, `Multiply`, and `Divide` to be variadic, too.

For example, calling `DivideMany(12, 4, 3)` should divide 12 by 4, divide the result of that by 3, then return *that* result. Calling `MultiplyMany` with ten inputs should return the result of multiplying them all together.

Once you're done, or if you need some hints, have a look at my suggested solution in Listing 8.3.

Takeaways

- To declare a function, we begin with the `func` keyword, followed by the name of the function, a list of parameters, and a list of results.
- The parameter list gives the *name* and *type* of each parameter that the function receives.
- Similarly, the result list gives the types (and, optionally, the names) of the result values that the function returns, if any.
- This is followed by the function *body*, a list of statements enclosed in curly braces.
- Simply *declaring* a function doesn't actually do anything, except notify Go that it exists: in order to execute the function's code, we need to *call* it.
- A function call consists of the name of the function, followed by a list of arguments in parentheses (if the function takes no parameters, the parentheses are empty).
- We can use the results returned by a function call, if any, as part of an assignment.
- Inside a function body, we can exit the function early using a `return` statement; if the function returns results, we have to give a list of values with `return`, one for each declared result.
- Functions *themselves* are values in Go, so we can assign them to variables, or pass them as arguments to other functions, or even return them as results.
- When using a function as a value, we don't necessarily have to give it a name; instead, we can write a *function literal*.
- A function literal can "see" all the variables in the enclosing function where it's defined, thus making it what's called a *closure* over them.
- This is powerful, since we can create a closure that executes later, but in this case we need to watch out for the enclosed variables being changed in the meantime—for example, in a loop.
- There are certain kinds of resources that can't be automatically cleaned up by the Go garbage collector, such as file handles, which need to be explicitly closed by calling `Close`.

- We can try to remember to explicitly clean these up before any exit point from a function, but a better way is to use `defer`.
- The `defer` keyword lets us defer execution of some block of code until the current function exits: for example, cleaning up the resource.
- If we *stack* defers by using several `defer` statements, the deferred code will be run in reverse order on exit: the last thing deferred is the first thing to run.
- You can name a function's result parameters, and the main value of this is to tell users and readers what those results *signify*: for example, latitude and longitude, rather than just `float64, float64`.
- *Naked* returns (bare `return` statements with no arguments) are legal syntactically, and you'll even see them used in some old blog posts and tutorials.
- But there's no good reason to use a naked return, and it's always clearer to return some explicit value, such as `nil`.
- When a function has named results, you can modify them in a deferred closure: for example, to return some error encountered when trying to clean up resources.
- Just because a function has named results doesn't mean it *must* use a naked return: it *can*, but it shouldn't, for the reasons we've discussed.
- A *variadic* function can take any number of arguments (including zero), using the special `...` form, and its arguments will appear inside the function as successive elements of a slice.

14: Building blocks

Most of the Go code that you or I write will be in the form of importable *packages*: useful, reusable chunks of functionality that we or other people can build into our programs. So what's a *program*?

Ultimately, if our Go code is to be any use to the world, it will need to be *executed*, that is to say, some actual computer somewhere will start our program, follow its instructions all the way to the end, and then stop. (Or not. Some programs, like network servers, are designed to run forever, or if not forever, then at least for the foreseeable future.)

In this chapter, we'll see how that works, and we'll take a peek into the interesting world of *command-line tools* in Go.

Compilation

In order for your program to run there needs to be some *executable binary*: a file containing machine code that your computer's CPU can understand, in a special format defined by your operating system.

When you run the program, whether by typing its name on a command line in a terminal session, or invoking it through a graphical interface such as the macOS Finder, the operating system knows how to load this file into memory and start executing the machine code at the right point.

Go uses a program called a *compiler* (that's what gives you all the angry messages about undeclared variables and mismatched types). The compiler reads your Go source code and translates it directly into machine code, producing an executable binary that you can then run.

All you need to distribute to the computer that will run your program is the executable binary file. This makes it very convenient to build and distribute Go programs: there's only a single file to manage.

The `main` package

In order for a bunch of Go source code to produce an executable, there has to be at least some code in a special package named `main`. Since you can't have more than one package in the same folder (apart from test packages), you'll need to put your `main` package in a subfolder of your project. Because the `main` package produces a *command* (that is, an executable), it's conventional to name this folder `cmd` (short for 'command').

It's not enough just to declare a `main` package, though. Whatever your program does, it has to *start* somewhere, and the Go compiler needs to know where that should be. The way we tell it is by defining a special function, also named `main`.

The `main` function

The flow of control in a Go program begins at the beginning of the `main` function, and proceeds from there in the way that you're already familiar with from the previous chapters in this book.

If `main` calls other functions, such as standard library functions like `fmt.Println`, or functions in your other packages, those will be executed, and control will then return to `main`.

When (and if) execution reaches the end of `main`, the program stops. Simple! Of course, lots of interesting things may happen along the way, but that's the basic shape of your Go program's life cycle.

The `init` function

In fact, `main` isn't always the first function that runs. If there's a function named `init` present, that will run first. If your `main` package imports other packages, and *they* have `init` functions, they will also all be run before your `main` function starts.

I don't recommend that you use `init`. While it can be useful to be able to set up certain things before the start of the program, or to automatically run code whenever a package is imported, `init` is rather magical and non-obvious. You may not even know that a package has an `init` function, unless you go looking for it, so this makes it harder to follow the flow of control.

And, as you've learned already in this book, it's a good idea to make the flow of control in your programs as straightforward and obvious as possible. If you read other people's programs, which I recommend, you'll find that they don't always do this.

Alternatives to `init`

Fortunately, we never need `init`. There are better ways to achieve the same result. The most obvious one is to just do whatever you need to do at the start of `main`!

In an imported package, though (which is to say any package whose name is not `main`), you can't control which function is called first. So if you need to initialise some data structure, for example, before anything else will work, one way to do that is to use a `var` statement.

While almost all Go code is inside functions, as you learned in the first chapter of this book, we're allowed to place `var` statements outside a function (at package level). And since `var` statements can not only declare variables but also assign values to them, including values returned from function calls, you can use this to do your initialisation.

For example:

```
package stuff


var thing = initialiseThing()
```

This presumes that you've defined the `initialiseThing` function somewhere in this package, but that's fine. The point is that you can make sure that function runs before any other in the package, and you don't need to rely on the magical behaviour of a function named `init`. This is just plain old Go code that everyone can read and understand.

Building an executable

Let's write an executable program that has a `main` package and a `main` function, then. Create a new folder named `hello` (make sure it's not a subfolder of any of your other Go projects). Add a file named `main.go` in this folder (just as with any other package, it's conventional, but not actually necessary, to give the file the same name as the package it implements).

Add this to the file:

```
package main

import "fmt"

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

You now know that the `package main` declaration is essential if we want to produce an executable. And we also need a function `main`, or the executable wouldn't do anything.

As you may know, you can run this program directly from the command line by typing `go run main.go`. That's useful for testing or experimenting, but this time we're interested in producing a binary executable.

Remember earlier in this book when we built a binary for our calculator program, using the `go build` command? Let's try that with this program. Run the following command within the `hello` folder:

```
go build
```

The executable binary file

If there are no errors, this command won't produce any output, but you'll find that a new file has been created alongside your `main.go`, named `hello`.

Let's inspect it to see what kind of file it is:

```
file hello
```

```
hello: Mach-O 64-bit executable x86_64
```

You may see different output depending on your operating system, or CPU architecture, but that's the point: the Go compiler has produced an executable file in just the format that your system requires.

To run it, use the program name itself as the command:

```
./hello
```

```
Hello, world!
```

Success! This may not be the first time you've built and run a Go program, but now you know exactly what's happening, and why. The operating system loaded your binary `hello` file and started executing it at the machine code instruction representing the beginning of the `main` function. The flow of control proceeded all the way to the end of this function, and then the program exited.

The nice thing about having a binary file like this is that you can copy it to any other computer and run it there, with no other dependencies (provided that it has the same

operating system and CPU architecture). That computer doesn't need to have the Go tools installed, or, indeed, anything else at all. You could run this binary in a Docker container, or in Kubernetes, or in any other way, and all you need is one small file (this program takes up about 2 MiB on my system).

Cross-compilation

You might be wondering how to build executables for other operating systems, such as Linux or Windows, and other types of CPU, such as ARM64? Do you need to compile your program again on every system that you want to create executables for?

Happily, the answer is no. You can build an executable for any system you like, right here on your own machine. The target operating system for the `go build` command is controlled by the `GOOS` environment variable (pronounced, we are assured by the Go team, "goose"). The target CPU architecture is set by the `GOARCH` variable (pronounced "gorch").

Let's try an example:

```
GOOS=windows go build
```

This produces a file called `hello.exe`, which is what you'd expect for a Windows executable. But is it?

```
file hello.exe
```

```
hello.exe: PE32+ executable (console) x86-64 (stripped to external
PDB), for MS Windows
```

How about an executable for an ARM-based Linux system?

```
GOOS=linux GOARCH=arm go build
```

```
file hello
```

```
 hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
 statically linked, Go BuildID=xxx, not stripped
```

Very handy! You can get a list of all the GOOS and GOARCH combinations supported by your Go version by running the command `go tool dist list`.

Exiting

As you saw earlier in this chapter, a Go program automatically exits (that is, stops executing and returns control to the operating system) once it reaches the end of `main`. But what if you want to exit before that? Or what if you want to signal the operating system that there was some kind of run-time error with the program?

Go provides the `os.Exit` function for exactly this purpose. You need to pass it a parameter called the *exit code*, or *exit status*. This is just a number, and conventionally, a

zero exit status indicates that everything's fine, whereas a non-zero value would imply that some error happened.

Here's an example:

```
package main

import "os"

func main() {
    os.Exit(1)
}
```

If you run this program (either by running it directly with `go run`, or by building an executable with `go build` and then running that) you'll see that the shell reports:

```
exit status 1
```

You can also exit the program at any point by calling `log.Fatal` with a message, or by calling the built-in `panic` function. Both of these will immediately terminate the program with a non-zero exit status (and, in the case of `panic`, some information about where in the code the panic was triggered).

The problem with either `log.Fatal` or `panic` is that, if they're buried deeply in the code, it can be hard for a reader to see how and where the program can exit. A good rule of thumb, then, is to only exit the program from the `main` function. It brought you into the world, if you like, so it should be the one to see you out.

And on that note, it's nearly time to see ourselves out. It's been great fun working together and learning about Go, and I hope you feel the same. In the final chapter, though, let's just say a brief word about *how* we should write Go programs: ideally, with kindness, simplicity, and humility, and without striving.

Takeaways

- The *compiled* form of a Go program is a special kind of file called an *executable binary*, containing machine code for a specific type of CPU and operating system.
- To produce an executable binary, our program needs to contain a `main` package, with a special function named `main` that is the program's entry point when it's run.
- Though if a function named `init` is present, it's automatically executed even before `main`, perhaps to do some initialisation, for example.
- However, it's widely considered poor style to use `init`, as we can initialise variables in a less magical way using plain old `var` statements.

- The `go build` command compiles a Go program with a `main` package into an executable binary that we can run (or distribute to someone else so that they can run it).
- Conveniently, we can build an executable for any CPU architecture or operating system we want, not just the one we happen to be running `go build` on, using the `GOOS` and `GOARCH` environment variables.
- While the program exits automatically when the end of `main` is reached, it can also exit at any other time by calling the `os.Exit` function (for example, to set an exit status code).
- Other ways to exit the program, which are not recommended, are calling `log.Fatal` or `panic`; in general, though, please *don't* panic.

15. The Tao of Go

> *You can make buffalo go anywhere, just so long as they want to go there.*
> —Jerry Weinberg, "Secrets of Consulting"

Tao refers to the inner nature or natural tendency of things. For example, water tends to flow downhill: that is its Tao. You can dam it, channel it, pump it, or otherwise interfere with it, but despite all your efforts, it will probably end up where it was going anyway.

To follow Tao, as a way of engaging with the world, is to be sensitive to the natural tendency of things, and not to waste energy struggling against them, but instead to go along with them: to work *with* the grain, not against it.

To extend the water analogy, a poor swimmer thrashes around making a lot of noise and fuss, but little actual progress. A Taoist, on the other hand, surfs.

So what is the Tao of Go? If we were to approach software development in Go in a sensitive, intelligent way, following the natural contours of the language and the problem rather than trying to bulldoze them out of the way, what would that look like? Let's try to establish a few general principles.

Kindness

> *Here are my three treasures; look after them! The first is kindness; the second, simplicity; the third, humility.*
> —"The Dàodé Jīng"

What does it mean to write programs with kindness and compassion? It means that we write code for human beings, not for computers. Those humans are fallible, impatient, inexperienced, inattentive, and in every other way imperfect. We can make their lives

(and jobs) considerably easier by putting some thought into the design and detail of our Go code.

We can be kind to our users by giving our packages descriptive names, by making them easy to import, by documenting them well, and by assigning them generous open-source licenses. We can design deep abstractions that let users leverage small, simple APIs to access powerful, useful behaviours.

We can be kind to those who run our programs by making them easy to install and update, by requiring a minimum of configuration and dependencies, and by catching the most common usage mistakes and runtime errors and giving the user helpful, accurate, and friendly information about what's wrong and how to fix it.

We can be kind to those who have to read our code by being as clear, as simple, and as explicit as possible. We can give types and functions meaningful names, that make sense in context, and let users create their own programs using our abstractions in straightforward, logical combinations. We can eliminate cognitive roadblocks and speed bumps by sticking to conventions, implementing standard interfaces, and doing the obvious thing the obvious way.

Finally, we can be kind to ourselves, by writing great tests, in the way that you've learned to do in this book. Tests make it easy to understand, fix, and improve our programs in the future. Without them, things can go downhill fast.

> *Once a code base turns to spaghetti, it is nearly impossible to fix.*
> —John Ousterhout, "A Philosophy of Software Design"

Another way to be kind to our future selves, and our successors, is to make small, continual improvements to the program's architecture and overall design. Good programs live a long time (some bad ones, too), and the cumulative effect of many little changes is usually to make the codebase messy, complicated, and kludgy. We can help to avoid this by taking a little extra time, whenever we visit the codebase on some errand, to refactor and clean it up. Since we rarely get the chance to rewrite the system from scratch, investing small amounts of time in micro-improvements over a long time is the only practical way to keep it healthy.

Simplicity

The second virtue the Tao teaches us is frugality, modesty, simplicity: doing a lot with a little, and eliminating clutter. Go itself is a frugal language, with minimal syntax and surface area. It doesn't try to do everything, or please everyone.

> *Our life is frittered away by detail. Simplify, simplify!*
> —Henry David Thoreau, "Walden"

We should do the same, by making our programs small and focused, uncluttered, doing one thing well. Deep abstractions provide a simple interface to powerful machinery. We don't make users do lots of paperwork in order to earn the privilege of calling our

package. Wherever we can provide a simple API with sensible defaults for the most common cases, we do so.

Flexibility is a good thing, but we shouldn't try to handle every case, or provide for every feature. Extensibility is great, but we shouldn't compromise a simple design to allow for things we don't need yet. Indeed, a simple program is easier to extend than a complicated one.

Humility

The third treasure is humility. Like water, the Taoist seeks the low places, without striving, competing, or attempting to impress others. Go itself is humble and pragmatic: it doesn't have all the high-tech features and theoretical advantages of some other languages. Indeed, it deliberately leaves out many features that are big selling points of other languages. Its designers were less interested in creating an impressive programming language, or in topping popularity polls, than in giving people a small and simple tool for getting useful work done in the most practical and direct way possible.

Go recognises that we are prone to mistakes, and it has many ways of protecting us from them. It takes care of allocating memory, cleaning up things we've finished with, and warns us about unused imports or variables. It's a language designed for people who know they don't know everything, and who understand their own propensity for mistakes: humble people, in other words.

> *The most dangerous error is failure to recognize our own tendency to error.*
> —Basil Liddell Hart, "Why Don't We Learn From History?"

As Go programmers, we can be humble by not trying to be too clever. We are not writing code to impress everybody with what terrific programmers we are: instead, we are content to do the obvious thing. We express ourselves clearly and straightforwardly, without feeling the need to obtrude our own personality on the code.

We use the standard library, when it solves the problem, and third-party packages only when it doesn't. If there's a de facto standard package for something, we use that: if it's good enough for others, it's good enough for us.

We can recognise that we don't know everything, and we can't make very accurate predictions (especially about the future), so we shouldn't waste time and energy pre-engineering things we may never need. We don't assume we know best what *other* software people will want to use in conjunction with ours, so we don't hard-wire in dependencies on it.

We assume that anything we write will contain bugs, so we write careful tests that try to elicit unexpected behaviour or incorrect results. We understand there will inevitably be important things we don't know or can't anticipate correctly, so we don't optimise code too much for the status quo, because a lot of that work will end up being wasted.

We are humble enough to review our *own* code before asking anyone else to, because if we can't be bothered, why should they? We take the time to go through it line by line,

reading it as a new user or developer would, trying to follow the argument logically. Is it clear where to start reading? Does the program introduce the crucial types or constants at the beginning, and go on to show how they're used? Do the names of things identify clearly and accurately what they do, or have they grown confusing and out of date through a dozen refactorings? Does the program fit neatly and naturally into its structure, or has it overgrown some parts and left others oddly empty?

Because we're not bound up with our own cleverness and elegance, we don't need to cram three or four different ideas into a single line of code. Instead, we set out the logic plainly, obviously, step by step, statement by statement, in bite-size pieces that do exactly the necessary thing in exactly the way that the reader expects. Where this isn't possible, we take the trouble to explain to the reader just what they need to know to understand what's going on.

Because we know we're not geniuses, and we can't write programs that are so brilliant as to be self-explanatory, we take some trouble over our explanations. We accompany code with documentation that shows not just what the *program* does, but how to accomplish things with it that *users* might want to do. We include detailed usage examples showing exactly what needs to be done, from scratch, to carry out realistic tasks, what users should expect to see when it's done and what they should do next, and we check those examples rigorously and regularly to make sure they *still work*.

Not striving

The final teaching of the Tao is *wúwéi* ("not striving"). This is sometimes misunderstood as laziness, withdrawal, or passivity; quite the opposite. Working hard does not always mean working well. We all know people who are chronically busy, always in a rush, fussing and flapping and in a fury of activity, but they never seem to actually achieve anything very much. Also, they're having a miserable time, because they know that too.

Instead, we often do our best work when it might seem to others that we're doing nothing at all: walking by a river on a beautiful day, or sitting on the porch watching a spider building a web. If we can be smart enough to stop *trying* so hard for just a minute, the right idea often pops into our heads straight away.

Rather than treating every problem as an enemy to be attacked, a mountain to be climbed, or a wall to be demolished, we can use the "not striving" principle (sometimes "not *forcing*" would be a better translation). We've probably all had the embarrassing experience of fruitlessly pushing on a stubborn door, only to realise that this particular door responds better to a *pull*. What little signs are we overlooking in our daily work that we should be pulling instead of pushing?

A problem-solving mindset is good, but problem *eliminating* is even better. Ask: how could we reframe this problem so that it just goes away? What restatement of the requirements would make the solution trivial and even obvious? Is there a simple and elegant design we're not seeing because we're fixated on some detail that turns out to be irrelevant? Could we get away without even trying to solve this problem? The best

optimisation is not to do the thing at all.

It's a common mistake to confuse programming with typing. If someone's just sitting there staring into space, it doesn't look like they're doing anything useful. If they're rattling furiously on a keyboard, though, we assume they're achieving something. In fact, real programming happens before the typing, and sometimes instead of it. When we've done a really good bit of programming, often the only key we need to press is the delete key.

Next time you hit a problem, try not striving or forcing things for once, and see if the problem can be gently encouraged to solve itself. If you find yourself struggling to get a buffalo to where you want it to go, stop struggling. Ask yourself if you can find out where the *buffalo* wants to go, and whether perhaps that isn't the best place for it after all.

The love of Go

Thanks for reading this book (and making it to the end!) You're well on your way to becoming a master software engineer, and I feel honoured to have shared this part of your journey with you. I wish you every success and happiness in the future... for the love of Go!

About this book

Who wrote this?

John Arundel is a Go teacher and mentor of many years experience. He's helped literally thousands of people to learn Go, with friendly, supportive, professional mentoring, and he can help you too. Find out more:

- Learn Go remotely with me

Feedback

If you enjoyed this book, let me know! Email go@bitfieldconsulting.com with your comments. If you didn't enjoy it, or found a problem, I'd like to hear that too. All your feedback will go to improving the book.

Also, please tell your friends, or post about the book on social media. I'm not a global mega-corporation, and I don't have a publisher or a marketing budget: I write and produce these books myself, at home, in my spare time. I'm not doing this for the money: I'm doing it so that I can help bring the love of Go to as many people as possible.

That's where you can help, too. If you love Go, tell a friend about this book!

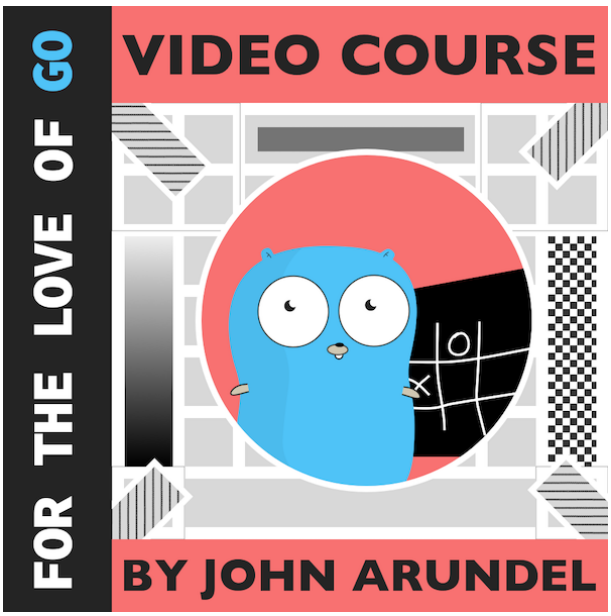
Mailing list

If you'd like to hear about it first when I publish new books, or even join my exclusive group of beta readers to give feedback on drafts in progress, you can subscribe to my mailing list here:

- Subscribe to Bitfield updates

Video course

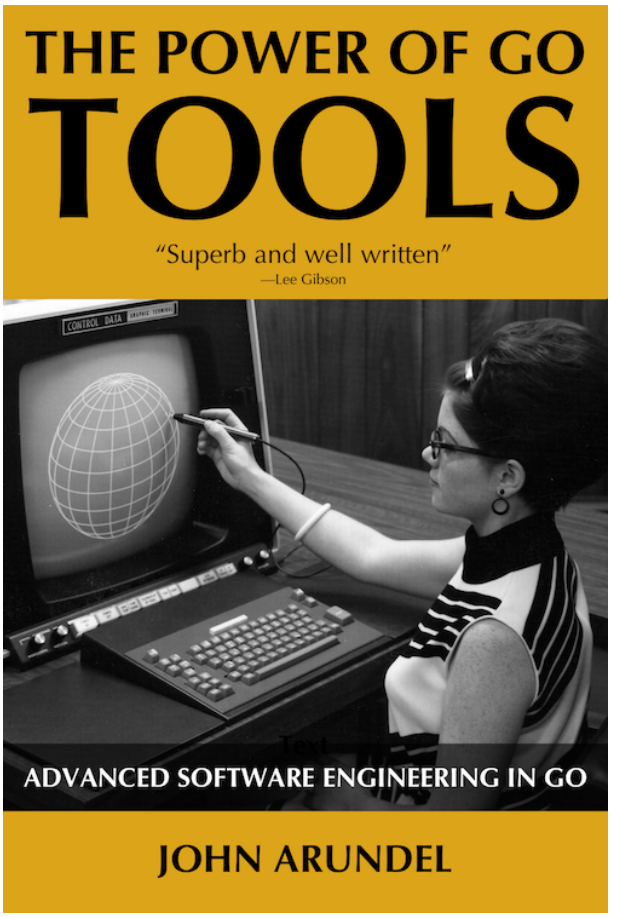
If you're one of the many people who enjoys learning from videos, as well as from books, you may like the *For the Love of Go* video course that accompanies this series:



- For the Love of Go: Video Course

The Power of Go: Tools

Now that you've finished this book, you're ready to unlock the power of Go, master obviousness-oriented programming, and learn the secrets of Zen mountaineering, with The Power of Go: Tools.



The Power of Go: Tools is the next step on your software engineering journey, explaining how to write simple, powerful, idiomatic, and even beautiful programs in Go.

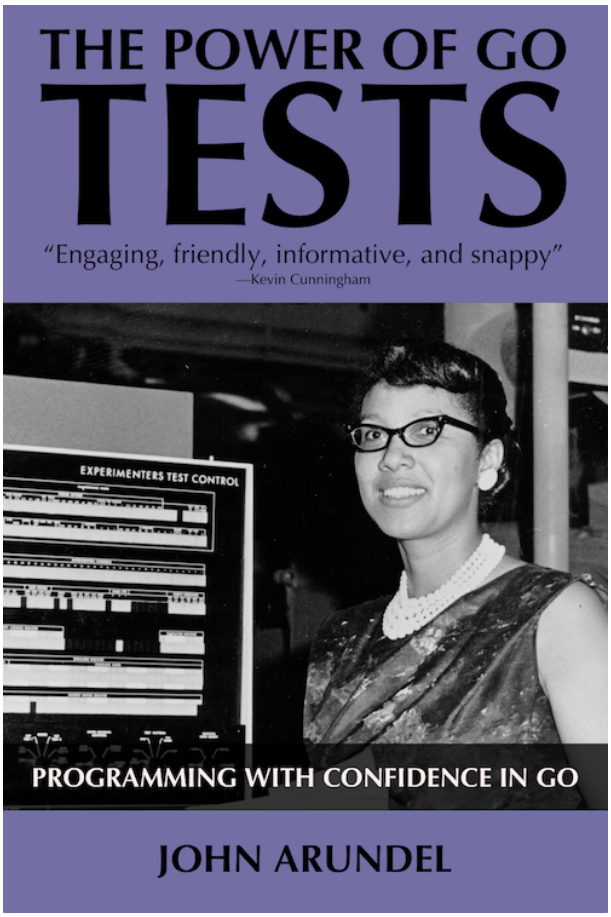
This friendly, supportive, yet challenging book will show you how master software engineers think, and guide you through the process of designing production-ready command-line tools in Go step by step.

How do you break down a problem into manageable chunks? How do you test functions before you've written them? How do you design reusable packages and tools that delight users? *The Power of Go: Tools* has the answers.

See the end of this book for a sneak preview of the first chapter of *The Power of Go: Tools*!

The Power of Go: Tests

What does it mean to program with confidence? How do you build self-testing software? What even is a test, anyway? The Power of Go: Tests answers these questions, and many more.



Welcome to the thrilling world of fuzzy mutants and spies, guerilla testing, mocks and crocks, design smells, mirage tests, deep abstractions, exploding pointers, sentinels and six-year-old astronauts, coverage ratchets and golden files, singletons and walking skeletons, canaries and smelly suites, flaky tests and concurrent callbacks, fakes, CRUD methods, infinite defects, brittle tests, wibbly-wobby timey-wimey stuff, adapters and ambassadors, tests that fail only at midnight, and gremlins that steal from the player during the hours of darkness.

> *If you get fired as a result of applying the advice in this book, then that's probably for the best, all things considered. But if it happens, I'll make it my personal mission to get you a job with a better company: one where people are rewarded, not*

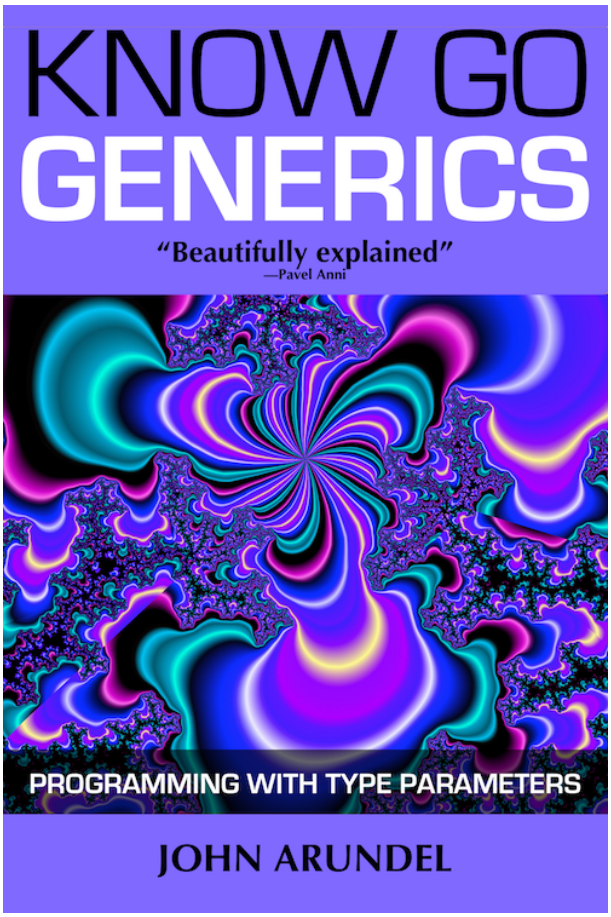
punished, for producing software that actually works.

Go's built-in support for testing puts tests front and centre of any software project, from command-line tools to sophisticated backend servers and APIs. This accessible, amusing book will introduce you to all Go's testing facilities, show you how to use them to write tests for the trickiest things, and distils the collected wisdom of the Go community on best practices for testing Go programs. Crammed with hundreds of code examples, the book uses real tests and real problems to show you exactly what to do, step by step.

You'll learn how to use tests to design programs that solve user problems, how to build reliable codebases on solid foundations, and how tests can help you tackle horrible, bug-riddled legacy codebases and make them a nicer place to live. From choosing informative, behaviour-focused names for your tests to clever, powerful techniques for managing test dependencies like databases and concurrent servers, The Power of Go: Tests has everything you need to master the art of testing in Go.

Know Go: Generics

Things are changing! Go beyond the basics, and master the new generics features introduced in Go 1.18. Learn all about type parameters and constraints in Go and how to use them, with this easy-to-read but comprehensive guide.



If you're new to Go and generics, and wondering what all the fuss is about, this book is for you! If you have some experience with Go already, but want to learn about the new generics features, this book is also for you. And if you've been waiting impatiently for Go to just get generics already so you can use it, don't worry: this book is for you too!

You don't need an advanced degree in computer science or tons of programming experience. Know Go: Generics explains what you need to know in plain, ordinary language, with simple examples that will show you what's new, how the language changes will affect you, and exactly how to use generics in your own programs and packages.

As you'd expect from the author of For the Love of Go and The Power of Go: Tools, it's fun and easy reading, but it's also packed with powerful ideas, concepts, and techniques that you can use in real-world applications.

Further reading

You can find more of my books on Go here:

- Go books by John Arundel

You can find more Go tutorials and exercises here:

- Go tutorials from Bitfield

I have a YouTube channel where I post occasional videos on Go, and there are also some curated playlists of what I judge to be the very best Go talks and tutorials available, here:

- Bitfield Consulting on YouTube

Credits

Gopher images by the magnificent egonelbre and MariaLetta.

Acknowledgements

It takes a village to raise a book, and the ideas in this one have been shaped and refined over several years in collaboration with my students at the Bitfield Institute of Technology, my online software engineering school. They have taught me just as much as I've taught them, and many have gone on to become amazing engineers at top-tier companies like Cisco, Microsoft, and Facebook. I feel very privileged to have had some involvement, however minor, with their craft mastery and career success.

I want to thank specifically, if not exclusively, Adebayo Ogidiolou, Bryan Green, Bukola Jimoh, Christian Sage, Dave Cottlehuber, Ivan Fetch, Jackson Kato, Jakub Jarosz, Jon Barber, Josh Akeman, Joumana El Alaoui, Kevin Strong, Mike Stanton, Parish Mwangi, Piero Mamberti, Quinn Murphy, Richard Bright, Sherif Abdalla, Thiago Nache, Thibaut Girier, Thom Kling, and Umang Vanjara. It's been my pleasure and honour to be part of your Go journey.

Many thanks also to the hundreds of beta readers who willingly read early drafts of various editions of this book, and took the time to give me detailed and useful feedback on where it could be improved. They never fail to help me identify explanations that don't work, jokes that don't land, code that doesn't compile, and (especially) things that make it harder for non-native English speakers to understand what I'm driving at. This book (and every other book I've written) is much, much better as a result.

If you'd like to be one of my beta readers in future, please go to my website, enter your email address, and tick the appropriate box to join my mailing list:

- https://bitfieldconsulting.com/

This book is dedicated to my mother, to whom I'm immensely grateful for the dozens of hours she spent listening patiently to my wild ideas, and helping me clarify and organise them into something resembling a book. Oh, and for giving me life in the first place. That helped too, obviously.

A sneak preview

If you're wondering where to go next after reading this book, here comes a sneak preview of the first chapter of The Power of Go: Tools. If you enjoy this chapter, please use the discount code `LOVE-TOOLS` to get 25% off the price of the full book. Just go to the product page, add the book to your cart, and then enter the code at the checkout.

Happy fun reading!

1. Packages

> *People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of stones.*
> —Donald Knuth

There are many ways to write books on programming, but my favourite is the one adopted by Brian Kernighan and Dennis Ritchie in their 1978 classic, 'The C Programming Language' (known as 'K&R', after the authors' initials):

> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages: print the words "Hello, world".

Why bother doing something so trivial? They explain:

> This is the basic hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

Now, this book is for Go programmers with a little experience, so I'll assume you're already capable of writing a "Hello, world" program in Go, compiling it, running it, and so on. (If not, I recommend you first read For the Love of Go, by the same author.)

Assuming at least a little experience with the Go language, then, this book will focus

on what to *do* with it. In other words, how do we build reliable, usable, well-engineered software with Go?

This will involve some ideas, techniques, and processes that may be new to you. It makes sense that when you're trying to learn something new, you should start from something that you already understand pretty well, such as a "Hello, World" program.

So let's start with a straightforward and (hopefully) familiar "Hello, world" program in Go, and develop it from there.

Hello, world

```
package main

import (
    "fmt"
)

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

(Listing 1.1)

Everything in Listing 1.1 should be familiar to experienced Gophers. We declare the package `main`, which has a special meaning to the Go compiler: it instructs it to build an executable binary.

We also declare the special *function* `main`, which will be the entry point to our binary. When the binary is run, `main` will be called automatically to start the program.

What does `main` do? It calls the standard library function `fmt.Println`, which prints a string to the program's standard output.

The name of this source file doesn't matter, since Go looks only at the `package` declaration; conventionally, though, code in package `main` is named `main.go`.

Let's run it:

```
go run main.go
```

```
Hello, world
```

This is fine: the program compiles and runs, and prints the expected message. But can we do better? What's wrong with this program, if anything?

Write packages, not programs

There's nothing wrong with it, in fact: many programs are designed to be run directly from a command line, to do something, and then to exit. Such programs are often referred to as *tools*.

Possibly one of the most significant inventions in computing, after the programming language itself, is the idea of an importable *package*: a re-usable unit of software that can be included in many different programs, usually to do some specific task.

Without packages we would always have to instruct the computer about every detail of what we want to do. The packages in the Go standard library, indeed, would be very hard to manage without. Every time we wanted to write an HTTP server, for example, we'd have to implement the HTTP protocol ourselves, which is no mean feat.

Go has excellent support for creating and using packages. Almost every Go project in existence is itself a package (or a library of related packages, like the standard library) that can be imported by other programs. This is a huge force multiplier for programmers: whatever you want to do, there's probably some package already written that does pretty nearly what you want.

In many cases, all we need to do to create a particular program is to figure out how to glue together the various packages that we need, and the job is done. We can then make *our* program a package, for use in turn by other programmers.

It follows, then, that any non-trivial code in your program should not be in package `main`; it should be in some *importable* package. All you need to do then is import that package and call it from your `main` function. And because you import it, others can import it and use it in their own programs, which they couldn't do if you had put it in package `main`.

Another reason to avoid putting code in `main` is that it's hard to test, for the same reason: you can't import it. So let's adopt this mindset from the start:

> *Write packages, not programs.*

Let's try to turn our "Hello, world" program from Listing 1.1 into a package. What will that involve?

Designing for a million users

The first step in designing a good package is a shift in mindset. This is no longer a tool just for our own private use or amusement: it's a public resource, an open-source asset that may be useful to thousands or even millions of other programmers.

We are now writing a piece of software that may be run in production at huge scale, in critical applications like healthcare, public transportation, or finance. It may stay in use for years or even decades, and be updated many times for security, language improvements, performance, new features, or other reasons.

People's livelihoods, or even their lives, may well depend upon it, and even if that's not true of our little "Hello, world" package, let's proceed as though it were, and build accordingly. If we treat every package we write as though it will have a million users, we won't go far wrong.

So what do we need? We need tests, we need a well-designed API, and we need documentation. Let's deal with testing first, for reasons that will become clear shortly.

Testing

Automated tests in Go can do anything you want, including nothing at all. What they usually do is import some package and call functions in it, comparing the results against some predetermined expectations.

This already gives us some important information about testing command-line interface (CLI) tools such as the "Hello, world" example. We will not be able to run the command itself as part of the test (or, at least, while we technically can do this, it's inconvenient and unnecessary).

Instead, we need a package that we can import, and some function in it that we can call. Since we can't import the `main` package, that implies we'll need to create some other package.

This isn't a problem, because we already decided that we were going to turn our tool into a package. But since we need a `package main` in order to build a binary, how do we reconcile this with all the actual functionality being in some non-`main` package?

The answer is: we import it. In fact, if we're lucky, `main` will need to do almost nothing but import our package and call some entrypoint function in it. This is ideal, as then there's no behaviour in `main` itself, so we don't need to test it, which is just as well, because we can't.

The engine package

The package that does all the work is what we may call an *engine* package. There are many parts to a car, but it's the engine that makes it move. Similarly, a Go application or module may have many packages, but there should be one central engine package that contains the core functionality. This is what the whole package is about; it may contain other packages, but they exist only to service and support the engine package.

There are some important decisions to make about the engine package: what should it be called? What functions and types should it have? Should it define some constants, and what should they be? Should it contain any global state?

While we could make all these decisions just by sitting down and *thinking* about them, I'm going to suggest a different approach.

We could argue all day about the design of the engine package, and many people are happy to do this, at least on company time. But even after a lot of discussion, we might still find that when we come to actually *use* the package, its API is not ideal. There may be important things missing. There may be things present that it turns out we don't need. Things we do need may have names and APIs that make sense in isolation, but not when used in context. And so on.

Instead, let's use a kind of mental trick. There's a Zen saying I'm fond of:

> *If you want to climb a mountain, begin at the top.*

Like most Zen sayings, this sounds (and is) absurd, but aims to lead us to something meaningful. If we usually only discover that the API is wrong when we come to call it, let's *start by calling it*, and save one entire wasted design cycle. We will write calling code that is sensible, readable, and useful, and only then proceed to implement everything necessary to make it work.

Let's imagine that the engine package already exists, then, and start using it. For example, let's try to write a test. What should we write?

The first test

A great principle to keep in mind with TDD is:

> *Test behaviour, not functions.*

Since we don't actually *have* any functions yet, this is more or less forced on us anyway. What do we mean by "behaviour" here? Simply, what the program does in a given situation.

In testing, we're interested only in behaviours we can observe from the outside. There may be all sorts of important things happening inside the package that we're not aware of, but when we speak of behaviour in this context, we're referring to things that *users* of the package would see.

What behaviour does the final program need to have? It needs to print out "Hello, world" on the terminal. Fine. Let's try to write a test for this behaviour.

GOAL: Write a test for the "Hello, world" printing program. This might be harder than it first appears.

Where should we put this test? Well, it needs to be in a file whose name ends with `_test.go`, and it will be testing a package whose name we might start by guessing will be `hello`. So let's call the file `hello_test.go` and start there.

Here's a first attempt:

```
package hello_test

import (
```

```
    "hello"
    "testing"
)

func TestPrintsHelloMessageToTerminal(t *testing.T) {
    t.Parallel()
    hello.Print()
}
```

Let's break this down, line by line. Since every Go file starts with a `package` clause, and the logic of what we're doing suggests `hello_test` would be a good name.

We need to import the standard library `testing` package for tests, and we will also need our engine package `hello`.

The test itself has a name, which can convey useful information if we choose it wisely. It's a good idea to name each test function after the behaviour it tests. You don't need to use words like `Should`, or `Must`; these are implicit. Just say what the function *does* when it's working correctly.

As you probably know, every Go test function takes a single parameter, conventionally named `t`, which is a pointer to a `testing.T` struct. This `t` value contains the state of the test during its execution, and we use its methods to control the outcome of the test (for example, calling `t.Error` to fail the test).

If you're not familiar with tests in Go, I recommend you read For the Love of Go, which will give you the background you'll need to make use of what follows.

Parallel tests

The first line of any test should be:

```
t.Parallel()
```

This signals that the test may be run concurrently with other tests, which is a good idea. Since we almost certainly have multiple CPUs available, it will be quicker to run multiple tests at once.

And there's another advantage to writing parallel tests. Even though our package may not itself use concurrency, it's entirely possible that *users* will want to call it concurrently. It's easy to overlook some race condition or global shared state when writing a package, so parallel tests are a way of battle-testing our code to make sure it's concurrency-safe.

If we do accidentally mutate some shared state somewhere, running all our tests in parallel gives us a good chance of catching this problem, especially if we run the tests with `go test -race` to enable the race detector.

> *Code in a package should always be safe to call concurrently. Parallel tests help ensure that.*

Occasionally there's a good reason why some test can't be run in parallel with others, but not often. In such cases, you can omit the call to `t.Parallel()`, and the test will be run on its own.

The first hard problem: naming things

In order to write this test, we needed to call the function, and in order to do that, we needed to decide its name. That, as I'm sure you already know, can often be tricky.

We know that the function will be in the `hello` package. What name would make sense? We might intuitively want to call it `Hello`, but since we're calling it from outside the package, we'll refer to it as `hello.Hello`, which is a stutter. Since the package name `hello` already tells us it's about saying hello, we don't need to repeat that word in the function name.

Since its job will be to print a message, perhaps `Print` would work. Complete with its package prefix, this will be referred to as `hello.Print`, which doesn't seem unreasonable. We'll see how it goes.

We can also ask at this point whether the `Print` function needs to take any parameters. Well, nothing springs to mind, so let's proceed to the next question: should it return any results?

Again, there doesn't seem to be anything obvious for the function to return, so let's assume it doesn't. So, assuming we've called the function successfully, we can ask: under what circumstances should the test fail?

Tests that pass silently, but fail loudly

But here we run into a problem: there don't appear to *be* any! The behaviour we want is that the function prints a message to the terminal, but how can a *test* verify that? It can't. We don't have access to the user's terminal from within the Go program.

There's another problem: running `go test` shouldn't cause anything to be printed out, unless there's a test failure. It's a good general design principle for command-line tools that they should not be too chatty in operation, and this is certainly the way that `go test` works. This is sometimes called "the rule of silence":

> *When a program has nothing surprising, interesting or useful to say, it should say nothing.*
> —http://www.linfo.org/rule_of_silence.html

But if the notional `Print` function does its job, it will print something! How can we solve this? When you frame the question this way, it's not hard to come up with the answer. Instead of having `Print` write to the terminal, give it something else to write to.

Faking the terminal

What do we really mean by "terminal" in this context, anyway? We mean `os.Stdout`. So now we can imagine that we pass something to the `Print` function that, in the real program, will actually *be* `os.Stdout`, but in the test, could be something else. What?

We could use an `*os.File`, which is what `os.Stdout` actually is, but let's think more generally. All we really care about is that we can *write* to whatever it is, and this idea is expressed by the standard library interface `io.Writer`. So let's make the parameter to `Print` be an `io.Writer`.

Fine. What concrete type that satisfies `io.Writer` shall we use in the test, though? We want a type that doesn't cause anything to be printed on the terminal, and ideally that we can inspect programmatically to see whether or not the expected message was actually written to it.

A `*bytes.Buffer` is ideal for this purpose, so let's create one, and pass it to the `Print` function. Here's the revised test:

```
func TestPrintsHelloMessageToWriter(t *testing.T) {
    t.Parallel()
    fakeTerminal := &bytes.Buffer{}
    hello.PrintTo(fakeTerminal)
    want := "Hello, world"
    got := fakeTerminal.String()
    if want != got {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

(Listing 1.2)

As you can see, the `Print` function now takes an argument: the writer to write to. However, this creates a problem with the name: `Print` suggests that it prints whatever arguments you give it. But the argument we're giving is not something to print: it's something to be printed *to*. So let's rename the function `PrintTo`, as in "print to... this!"

The name of the test also needed to change, because the required behaviour now isn't "prints to terminal", but "prints to writer". And this is a reassuringly Go-like API: rather than limiting ourselves to some concrete type, we take an interface (`io.Writer`) to make our code more flexible and widely useful. This change is worth making in any case, but it's interesting that it was more or less forced on us just by trying to *test* the function.

And we should expect to change our minds like this many times during the process of building a package, as we work through the TDD process. It's not a sign that we did anything wrong: in fact, we did everything *right*. We proceeded step by step from our ana-

lysis of the required behaviour, and evolved our API design by *using* it to do something specific, rather than reasoning about it in the abstract.

So now we have something that we can actually test: we can look at `fakeTerminal` afterwards to see if it contains the expected message. We define a variable `want` with that message, read the contents of the buffer into a variable called `got`, and then compare the two.

The end of the beginning

It's worth reflecting on what's happened during this test-writing process, because we've done quite a bit of useful design work:

- We've decided the name of the engine package
- We've decided the name of the entrypoint function
- We've written down its desired behaviour in words
- We've *injected* an `io.Writer` to make the package testable, and to prevent unwanted terminal output

All this before we've written a single line of system code, and in fact, that makes perfect sense, because if we'd started writing code, we would have ended up having to change it anyway when we came to the test. We would have named the function `Hello`, which makes sense *inside* the package, but not outside. We would have probably printed directly to the terminal using `fmt.Println`, which seems sensible until you come to try to test it.

This is the secret power of test-driven development: by writing the test first, as though the function already existed, we can solve a number of design problems in advance, and we also avoid the trap of writing an untestable function.

We also get a test at the end of it, of course, which is great, and we should never underestimate the value of having tests (try working on a non-trivial codebase without them). While most people agree that tests are desirable, if they're not written *first*, then they often don't get written at all. It's very hard to add tests afterwards. It may even be impossible without significant refactoring, and who wants to refactor code without the safety net of tests?

Takeaways

- When we're trying to learn a new way of doing things, it's best to start from a place where we're already comfortable.
- Packages are a force multiplier, so we should always seek to write a package that can be used in many different programs, not just one.
- A good way to write better code is to adopt the mindset that we're writing programs for millions of users, rather than simply for ourselves.

- Since no one can import, or test, your `main` package, it follows that all non-trivial behaviour should be implemented in a importable package instead.
- A Go program may contain many packages, but there's usually one central *engine* package that provides the main behaviour users care about.
- The best way to design a good package API is, slightly counter-intuitively, to start by *using* it—in a test, for example.
- The important thing to test is *behaviour*, not functions, and this is especially relevant to command-line tools.
- The names of test functions can contain useful information, if you choose to put it there.
- Parallel tests are not only faster, but help to flush out potential concurrency bugs, even when the package doesn't use concurrency itself.
- Functions that print output should take an `io.Writer` to print to, so that we can avoid noise when running tests, and also capture the output to check it.
- Tests are fundamentally about comparing `want` and `got`, whatever that means for the specific situation, so writing them this way helps people understand what your tests are looking for.
- Designing a package test-first seems hard at the beginning, or even a waste of time, but it gets us to a good design faster by eliminating a lot of rework.
- It's also nice to *have* tests.

Going further

If you're impatient to read more, by all means go on to the next chapter. But if you'd like to explore some of these ideas further, I have some suggestions for mini-projects that will give you a chance to practice applying TDD principles in Go.

1. If you've previously written a Go program where everything was in `package main`, try updating that program to use an engine package instead.

   Write tests for each of the important behaviours your program has, and try to organise the package code so that your `func main` is reduced to the absolute minimum.

2. Instead of a program that simply prints "Hello, world", try writing a program that asks the user for their name, then prints "Hello, [NAME]".

   What would its package API look like? How would you test it? What would its `main` function look like?

3. Try writing a program that prints the current time in the following format: `"It's X minutes past Y"`.

Supposing you have some function that does this, how can you test it when you don't know in advance exactly what time will be printed?

Your tests should produce no output when they pass (so they mustn't print the time to the user's terminal when they run `go test`, for example).