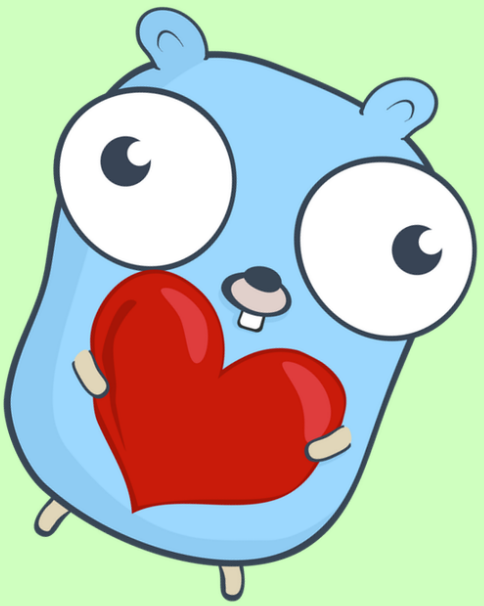
For the Love of Go (sample)

John Arundel

September 4, 2022

FOR THE LOVE OF GO



“Personable, human, and funny”

—Kevin Cunningham

JOHN ARUNDEL

Contents

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.