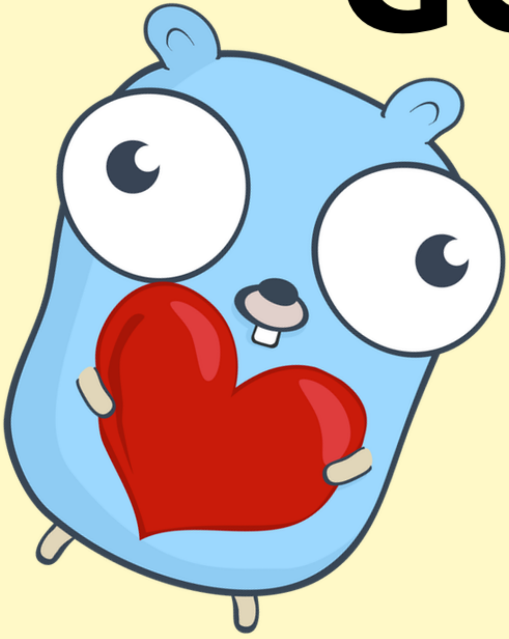
For the Love of Go (sample)

John Arundel

February 25, 2022

FOR THE LOVE OF GO



John Arundel
@bitfield

"A great way to dive into Go!"
—Max VelDink

Contents

9. 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`.

```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`:

```go
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"".

Recall from previous chapters that this value 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
```

You'd never make such a silly mistake, but in more complex programs, type errors can and do happen. 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 total 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 `:=`.

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`.