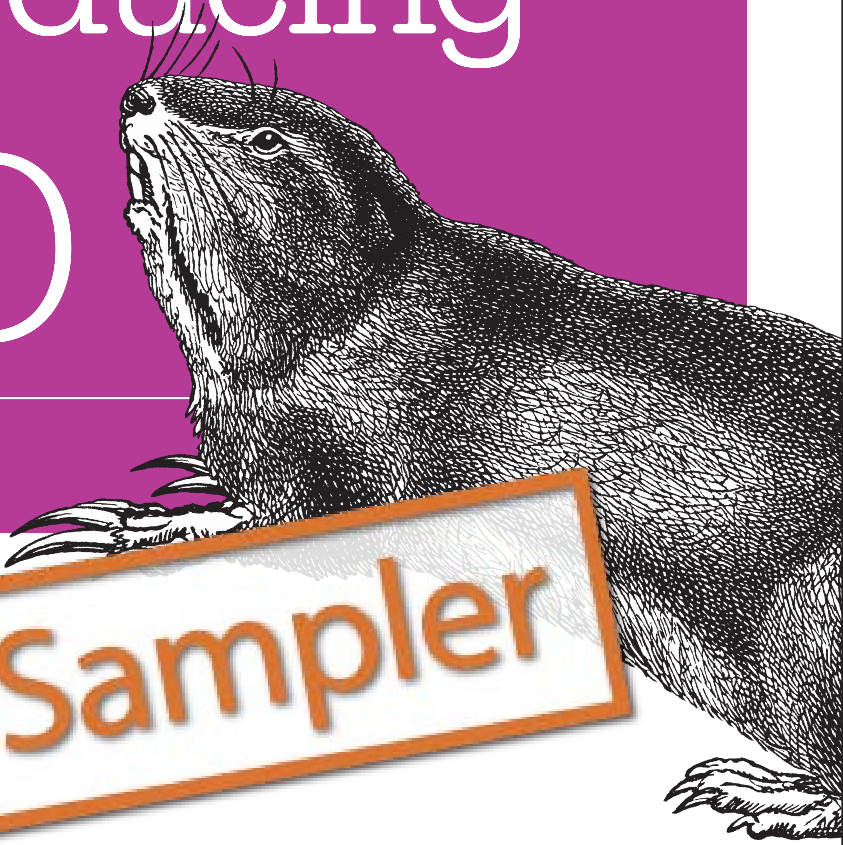
O'REILLY®

Introducing Go

BUILD RELIABLE, SCALABLE PROGRAMS



Free Sampler

Caleb Doxsey

Introducing Go
by Caleb Doxsey

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian MacDonald and Meghan Blanchette
Production Editor: Shiny Kalapurakkel
Copyeditor: Jasmine Kwityn
Proofreader: James Fraleigh
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January 2016: First Edition

Revision History for the First Edition
2016-01-06: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491941959* for release details.

978-1-491-94195-9

[LSI]

Table of Contents

CHAPTER 1

Getting Started

Go is a general-purpose programming language with advanced features and a clean syntax. Because of its wide availability on a variety of platforms, its robust well-documented common library, and its focus on good software engineering principles, Go is a great programming language to learn.

This book assumes no prior knowledge of Go, and is intended to serve as an easy introduction to the language. All of the language's core features will be covered in short, concise chapters that should prepare you to write real Go programs and tackle some of the more advanced resources available on the language (online documentation, books, talks, etc.).

Although this book is suitable for inexperienced programmers, if you have never programmed before you will probably find the material too difficult to follow. You may benefit from consulting a more general programming resource before diving into the material here, but in all honesty, most students need the kind of hands-on, personal support that you might find in a classroom setting or one on one with an experienced developer.

Machine Setup

This book contains many code samples and exercises. For best results, you should try to run these examples on your own computer as you work your way through each chapter.

But before you can write your first Go program, there are a few things you will need to set up.

Text Editors

Go is a very readable, succinct language and so any text editor will work for editing files. There are plug-ins that add a few helpful features (like autocomplete and format-on-save) for many popular editors, but those plug-ins are not necessary to learn the language. If you're not sure what to use, I recommend using GitHub's Atom —it's free, cross-platform, and easy to install from the Atom website.

The Terminal

Go is a compiled language, and like many languages, it makes heavy use of the command line. If you're coming from a language that does most things through an IDE (such as Java or C#), this may be a bit intimidating, but thankfully, the Go tools are fairly easy to use. As a reminder, here's how you can get to a terminal:

Windows
: On Windows, the terminal (also known as the command prompt) can be brought up by pressing the Windows key + R (hold down the Windows key, then press R), typing `cmd.exe`, and hitting Enter.

OS X
: On OS X, the terminal can be reached by navigating to Finder → Applications → Utilities → Terminal.

Environment

Environment variables are a mechanism provided by your operating system for altering the behavior of a program without having to change it. An environment is a collection of these variables, each of which has a name and a corresponding value. For example, there is a `TEMP` environment variable that stores the location of a directory on your computer where temporary files are stored.

The Go toolset uses an environment variable called `GOPATH` to find Go source code. Although you're welcome to set the `GOPATH` to anything you want, to make things easier we will set it to be the same as your home directory:

Windows
: On Windows, user information is typically stored in *C:\Users\<USERNAME>*, where *<USERNAME>* would be replaced with your username (e.g., *C:\Users\alice*). Windows comes with a predefined environment variable called `USERPROFILE`, which you can use to set your `GOPATH`.

  Open a new terminal window and enter the following:

```
setx GOPATH %USERPROFILE%
```

If you're using a version of Windows prior to Vista, this command may not work, so you can also set environment variables by navigating to Control Panel → System → Advanced → Environment Variables.

OS X

On OS X, user information is typically stored in */Users/<USERNAME>*, where *<USERNAME>* would be replaced with your username (e.g., `/Users/alice`). On OS X, we will set `GOPATH` using a special initialization file for the terminal called *.bash_profile*.

Open a terminal and enter the following:

```
echo 'export GOPATH=$HOME\n' >> ~/.bash_profile
```

Close the terminal, reopen it, and enter the following:

```
env
```

Among many other environment variables, you should see an entry for `GOPATH`.

Go

Go is both the name of the programming language and the name for the toolset used to build and interact with Go programs. Before you begin working with Go, you'll need to install the Go toolset.

Download and run the installer for your platform from *golang.org/dl*.

To confirm everything is working, open a terminal and type the following:

```
go version
```

You should see the following (your version number and operating system may be slightly different):

```
go version go1.5 windows/amd64
```

If you get an error about the command not being recognized, try restarting your computer.

The Go toolset is made up of several different commands and subcommands. You can pull up a list of those commands by typing:

```
go help
```

With Go installed and working, you now have everything you need to write your first Go program.

Your First Program

Traditionally, the first program you write in any programming language is called a "Hello, World" program—a program that simply outputs `Hello, World` to your terminal. Let's write one using Go.

First, create a new folder where you can store our "Hello, World" program. Create a folder named *~/src/golang-book/chapter1*. From the terminal, you can do this by entering the following commands:

On Windows

```
md src\golang-book\chapter1
```

On OS X

```
mkdir -p src/golang-book/chapter1
```

Open your text editor, create a new file, and enter the following:

```
package main

import "fmt"

// this is a comment

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

Make sure your file is identical to what is shown here and save it as *main.go* in the folder we just created. Open up a new terminal and type in the following:

```
cd src/golang-book/chapter1
go run main.go
```

You should see `Hello, World` displayed in your terminal. The `go run` command takes the subsequent files (separated by spaces), compiles them into an executable saved in a temporary directory, and then runs the program. If you didn't see `Hello, World` displayed, you may have made a mistake when typing in the program. The Go compiler will give you hints about where the mistake lies. Like most compilers, the Go compiler is extremely pedantic and has no tolerance for mistakes.

How to Read a Go Program

Let's look at this program in more detail:

```
package main

import "fmt"

// this is a comment

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

Go programs are read top to bottom, left to right (like a book). The first line says this:

```
package main
```

This is known as a *package declaration*, and every Go program must start with it. Packages are Go's way of organizing and reusing code. There are two types of Go programs: executables and libraries. Executable applications are the kinds of programs that we can run directly from the terminal (on Windows, they end with *.exe*). Libraries are collections of code that we package together so that we can use them in other programs. We will explore libraries in more detail later; for now, just make sure to include this line in any program you write.

The next line is blank. Computers represent newlines with a special character (or several characters). Newlines, spaces, and tabs are known as whitespace (because you can't see them). Go mostly doesn't care about whitespace—we use it to make programs easier to read (you could remove this line and the program would behave in exactly the same way).

On the following line, we see this:

```
import "fmt"
```

The `import` keyword is how we include code from other packages to use with our program. The `fmt` package (shorthand for format) implements formatting for input and output. Given what we just learned about packages, what do you think the `fmt` package's files would contain at the top of them?[1]

Notice that `fmt` is surrounded by double quotes. The use of double quotes like this is known as a *string literal*, which is a type of *expression*. In Go, strings represent a sequence of characters (letters, numbers, symbols, etc.) of a definite length. Strings are described in more detail in the next chapter, but for now the important thing to

1 Files in the `fmt` package start with `package fmt`.

keep in mind is that an opening " character must eventually be followed by a closing " character and anything in between the two is included in the string (the " character itself is not part of the string).

The line that starts with `//` is known as a *comment*. Comments are ignored by the Go compiler and are there for your own sake (or whoever picks up the source code for your program). Go supports two different styles of comments: `//` comments in which all the text between the `//` and the end of the line is part of the comment, and `/* */` comments where everything between the asterisks is part of the comment (and may include multiple lines).

After this, you see a function declaration:

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

Functions are the building blocks of a Go program. They have inputs, outputs, and a series of steps called statements that are executed in order. All functions start with the keyword `func` followed by the name of the function (`main`, in this case), a list of zero or more *parameters* surrounded by parentheses, an optional return type, and a *body* which is surrounded by curly braces. This function has no parameters, doesn't return anything, and has only one statement. The name `main` is special because it's the function that gets called when you execute the program.

The final piece of our program is this line:

```
fmt.Println("Hello, World")
```

This statement is made of three components. First, we access another function inside of the `fmt` package called `Println` (that's the `fmt.Println` piece); `Println` means "print line." Then we create a new string that contains `Hello, World` and *invoke* (also known as *call* or *execute*) that function with the string as the first and only argument.

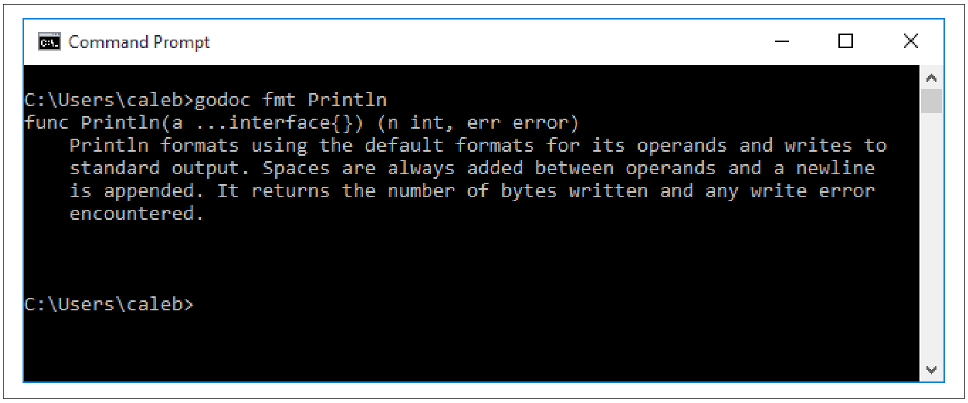
At this point, you've already seen a lot of new terminology. Sometimes it's helpful to deliberately read your program out loud. One reading of the program we just wrote might go like this:

> Create a new executable program that references the `fmt` library and contains one function called `main`. That function takes no arguments and doesn't return anything. It accesses the `Println` function contained inside of the `fmt` package and invokes it using one argument—the string `Hello, World`.

The `Println` function does the real work in this program. You can find out more about it by typing the following in your terminal:

```
godoc fmt Println
```

Among other things, you should see the output shown in Figure 1-1.



```
C:\Users\caleb>godoc fmt Println
func Println(a ...interface{}) (n int, err error)
    Println formats using the default formats for its operands and writes to
    standard output. Spaces are always added between operands and a newline
    is appended. It returns the number of bytes written and any write error
    encountered.


C:\Users\caleb>
```

Figure 1-1. Output of godoc fmt Println

Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

Go is a very well-documented programming language, but this documentation can be difficult to understand unless you are already familiar with programming languages. Nevertheless, the `godoc` command is extremely useful and a good place to start whenever you have a question.

Back to the function at hand, this documentation is telling you that the `Println` function will send whatever you give to it to *standard output* (i.e., the output of the terminal you are working in). This function is what causes `Hello, World` to be displayed.

In the next chapter, we will explore how Go stores and represents things like `Hello, World` by learning about types.

Exercises

1. What is whitespace?
2. What is a comment? What are the two ways of writing a comment?
3. Our program began with `package main`. What would the files in the `fmt` package begin with?
4. We used the `Println` function defined in the `fmt` package. If you wanted to use the `Exit` function from the `os` package, what would you need to do?
5. Modify the program we wrote so that instead of printing `Hello, World` it prints `Hello, my name is` followed by your name.