O'REILLY®

Second Edition

Learning Go

An Idiomatic Approach to Real-World Go Programming

Early Release
RAW & UNEDITED

Jon Bodner

O'REILLY®

Second Edition

Learning Go

An Idiomatic Approach to Real-World Go Programming

Early Release
RAW & UNEDITED

Jon Bodner

Learning Go

SECOND EDITION

An Idiomatic Approach to Real-World Go Programming

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jon Bodner

Beijing • Boston • Farnham • Sebastopol • Tokyo

Learning Go

by Jon Bodner

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://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Suzanne McQuade
- Developmental Editor: Rita Fernando
- Production Editor: Beth Kelly
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- March 2021: First Edition
- March 2024: Second Edition

Revision History for the Early Release

- 2022-10-26: First Release
- 2023-02-01: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492077213 for release details.

978-1-098-13923-0

[]

Preface

My first choice for a book title was *Boring Go* because, properly written, Go is boring.

It might seem a bit weird to write a book on a boring topic, so I should explain. Go has a small feature set that is out of step with most other modern programming languages. Well-written Go programs tend to be straightforward and sometimes a bit repetitive. There's no inheritance, no aspect-oriented programming, no function overloading, and certainly no operator overloading. There's no pattern matching, no named parameters, no exceptions. After a decade of debate and design, generics were added, but they have limitations that aren't present in other langauges. To the horror of many, Go has *pointers*. Go's concurrency model is unlike other languages, but it's based on ideas from the 1970s, as is the algorithm used for its garbage collector. In short, Go feels like a throwback. And that's the point.

Boring does not mean *trivial*. Using Go correctly requires an understanding of how its features are intended to fit together. While you can write Go code that looks like Java or Python, you're going to be unhappy with the result and wonder what all the fuss is about. That's where this book comes in. It walks through the features of Go, explaining how to best use them to write idiomatic code that can grow.

When it comes to building things that last, being boring is great. No one wants to be the first person to drive their car over a bridge built with untested techniques that the engineer thought were cool. The modern world depends on software as much as it depends on bridges, perhaps more so. Yet many programming languages add features without thinking about their impact on the maintainability of the codebase. Go is intended for building programs that last, programs that are modified by dozens of developers over dozens of years.

Go is boring and that's fantastic. I hope this book teaches you how to build exciting projects with boring code.

Who Should Read This Book

This book is targeted at developers who are looking to pick up a second (or fifth) language. The focus is on people who are new to Go. This ranges from those who don't know anything about Go other than it has a cute mascot, to those who have already worked through a Go tutorial or even written some Go code. The focus for *Learning Go* isn't just how to write programs in Go; it's how to write Go *idiomatically*. More experienced Go developers can find advice on how to best use the newer features of the language. The most important thing is that the reader wants to learn how to write Go code that looks like Go.

Experience is assumed with the tools of the developer trade, such as version control (preferably Git) and IDEs. Readers should be familiar with basic computer science concepts like concurrency and abstraction, as the book explains how they work in Go. Some of the code examples are downloadable from GitHub and dozens more can be tried out online on The Go Playground. While an internet connection isn't required, it is helpful when reviewing executable examples. Since Go is often used to build and call HTTP servers, some examples assume familiarity with basic HTTP concepts.

While most of Go's features are found in other languages, Go makes different tradeoffs, so programs written in it have a different structure. *Learning Go* starts by looking at how to set up a Go development environment, and then covers variables, types, control structures, and functions. If you are tempted to skip over this material, resist the urge and take a look. It is often the details that make your Go code idiomatic. Some of what seems obvious at first glance might actually be subtly surprising when you think about it in depth.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/learning-go-book*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Go* by Jon Bodner (O'Reilly). Copyright 2021 Jon Bodner, 978-1-492-07721-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/learn-go*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments

Writing a book seems like a solitary task, but it doesn't happen without the help of a great number of people. I mentioned to Carmen Andoh that I wanted to write a book on Go and at GopherCon 2019, and she introduced me to Zan McQuade at O'Reilly. Zan guided me through the acquisition process and continued to provide me advice while I was writing *Learning Go*. Michele Cronin edited the text, gave feedback, and listened during the inevitable rough patches. Tonya Trybula's copy editing and Beth Kelly's production editing made my draft production-quality.

While writing, I received critical feedback (and encouragement) from many people including Jonathan Altman, Jonathan Amsterdam, Johnny Ray Austin, Chris Fauerbach, Chris Hines, Bill Kennedy, Tony Nelson, Phil Pearl, Liz Rice, Aaron Schlesinger, Chris Stout, Kapil Thangavelu, Claire Trivisonno, Volker Uhrig, Jeff Wendling, and Kris Zaragoza. I'd especially like to recognize Rob Liebowitz, whose detailed notes and rapid responses made this book far better than it would have been without his efforts.

My family put up with me spending nights and weekends at the computer instead of with them. In particular, my wife Laura graciously pretended that I didn't wake her up when I'd come to bed at 1 A.M. or later.

Finally, I want to remember the two people who started me on this path four decades ago. The first is Paul Goldstein, the father of a childhood friend. In 1982, Paul showed us a Commodore PET, typed PRINT 2 + 2, and hit the enter key. I was amazed when the screen said 4 and was instantly hooked. He later taught me how to program and even let me borrow the PET for a few weeks. Second, I'd like to thank my mother for encouraging my interest in programming and computers, despite having no idea what any of it was for. She bought me the BASIC programming cartridge for the Atari 2600, a VIC-20 and then a Commodore 64, along with the programming books that inspired me to want to write my own someday.

Thank you all for helping make this dream of mine come true.

Chapter 1. Setting Up Your Go Environment

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Every programming language needs a development environment, and Go is no exception. If you've already built a Go program or two, then you have a working environment, but you might have missed out on some of the newer techniques and tools. If this is your first time setting up Go on your computer, don't worry; it's easy to install Go and its supporting tools. After setting up the environment and verifying it, we will build a simple program, learn about the different ways to build and run Go code, and then explore some tools and techniques that make Go development easier.

Installing the Go Tools

To build Go code, you need to download and install the Go development tools. The latest version of the tools can be found at the downloads page on the Go website. Choose the download for your platform and install it. The

.pkg installer for Mac and the *.msi* installer for Windows automatically install Go in the correct location, remove any old installations, and put the Go binary in the default executable path.

> **TIP**
>
> If you are a Mac developer, you can install Go using Homebrew with the command `brew install go`. Windows developers who use Chocolatey can install Go with the command `choco install golang`.

The various Linux and BSD installers are gzipped tar files and expand to a directory named *go*. Copy this directory to */usr/local* and add */usr/local/go/bin* to your `$PATH` so that the `go` command is accessible:

```
$ tar -C /usr/local -xzf go1.19.5.linux-amd64.tar.gz
$ echo 'export PATH=$PATH:/usr/local/go/bin' >>
$HOME/.bash_profile
$ source $HOME/.bash_profile
```

You might need root permissions to write to */usr/local* . If the `tar` command fails, rerun it with `sudo tar -C /usr/local -xzf go1.19.5.linux-amd64.tar.gz`.

> **NOTE**
>
> Go programs compile to a single native binary and do not require any additional software to be installed in order to run them. This is in contrast to languages like Java, Python, and JavaScript, which require you to install a virtual machine to run your program. Using a single native binary makes it a lot easier to distribute programs written in Go. This book doesn't cover containers, but developers who use Docker or Kubernetes can often package a Go app inside a scratch or distroless image. You can find details in this blog post by Geert Baeke.

You can validate that your environment is set up correctly by opening up a terminal or command prompt and typing:

```
$ go version
```

If everything is set up correctly, you should see something like this printed:

```
go version go1.19.5 darwin/arm64
```

This tells you that this is Go version 1.18.4 on macOS. (Darwin is the operating system at the heart of macOS and arm64 is the name for the 64-bit chips based on ARM's designs.) On x64 Linux, you would see:

```
go version go1.19.5 linux/amd64
```

Troubleshooting Your Go Installation

If you get an error instead of the version message, it's likely that you don't have `go` in your executable path, or you have another program named `go` in your path. On macOS and other Unix-like systems, use `which go` to see the `go` command being executed, if any. If nothing is returned, you need to fix your executable path.

If you're on Linux or BSD, it's possible you installed the 64-bit Go development tools on a 32-bit system or the development tools for the wrong chip architecture.

Go Tooling

All of the Go development tools are accessed via the `go` command. In addition to `go version`, there's a compiler (`go build`), code formatter (`go fmt`), dependency manager (`go mod`), test runner (`go test`), a tool that scans for common coding mistakes (`go vet`) and more. They are covered in detail in Chapter 4, Chapter 5, and Chapter 15. For now, let's take a quick look at the most commonly used tools by writing everyone's favorite first application: Hello World.

> **NOTE**
>
> Since the introduction of Go in 2009, there have been several changes in how Go developers organize their code and their dependencies. Because of this churn, there's lots of conflicting advice, and most of it is obsolete (for example, you can safely ignore discussions about GOROOT and GOPATH).
>
> For modern Go development, the rule is simple: you are free to organize your projects as you see fit and store them anywhere you want.

Your First Go Program

Let's go over the basics of writing a Go program. Along the way, we will see the parts that make up a simple Go program. You might not understand everything just yet, and that's OK; that's what the rest of the book is for!

Making a Go Module

The first thing we need to do is create a directory to hold our program. Call it *ch1*. On the command line, enter the new directory. If your computer's terminal uses bash or zsh, this looks like:

```
$ mkdir ch1
$ cd ch1
```

Inside the directory, run the `go mod init` command to mark this directory as a Go *module*:

```
$ go mod init hello_world
go: creating new go.mod: module hello_world
```

You will learn more about what a module is in Chapter 4, but for now, all you need to know is that a Go project is called a module. A module is not just source code. It is also an exact specification of the dependencies of the code within the module. Every module has a `go.mod` file in its root directory. Running `go mod init` creates this file for us. The contents of a basic `go.mod` file look like this:

```
module hello_world

go 1.19
```

The `go.mod` file declares the name of the module, the minimum supported version of Go for the module, and any other modules that your module depends on. You can think of it as being similar to the `requirements.txt` file used by Python or the `Gemfile` used by Ruby.

You shouldn't edit the `go.mod` file directly. Instead, use the `go get` and `go mod tidy` commands to manage changes to the file. Again, all things related to modules are covered in Chapter 4.

go build

Now let's write some code! Open up a text editor, enter the following text, and save it inside *ch1* with the file name *hello.go*:

```
package main

import "fmt"

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

(Note to everyone about to file an indentation errata: I meant to do that! You will see why in just a bit.)

Let's quickly go over the parts of the Go file we created. The first line is a package declaration. Within a Go module, code is organized into one or more packages. The `main` package in a Go module contains the code that starts a Go program.

Next there is an import declaration. The `import` statement lists the packages referenced in this file. We're using a function in the `fmt` (usually pronounced "fumpt") package from the standard library, so we list the package here. Unlike other languages, Go only imports whole packages.

You can't limit the import to specific types, functions, constants, or variables within a package.

All Go programs start from the `main` function in the `main` package. We declare this function with `func main()` and a left brace. Like Java, JavaScript, and C, Go uses braces to mark the start and end of code blocks.

The body of the function consists of a single line. It says that we are calling the `Println` function in the `fmt` package with the argument `"Hello, world!"`. As an experienced developer, you can likely guess what this function call does.

After the file is saved, go back to your terminal or command prompt and type:

```
$ go build
```

This creates an executable called `hello_world` (or *hello_world.exe* on Windows) in the current directory. Run it and you will unsurprisingly see `Hello, world!` printed on the screen:

```
$ ./hello_world
Hello, world!
```

The name of the binary matches the name in the module declaration. If you want a different name for your application, or if you want to store it in a different location, use the `-o` flag. For example, if we wanted to compile our code to a binary called "hello," we would use:

```
$ go build -o hello
```

In "Use go run to try out small programs", we will cover another way to execute a Go program.

go fmt

One of the chief design goals for Go was to create a language that allowed you to write code efficiently. This meant having simple syntax and a fast compiler. It also led Go's authors to reconsider code formatting. Most languages allow a great deal of flexibility on how code is formatted. Go does not. Enforcing a standard format makes it a great deal easier to write tools that manipulate source code. This simplifies the compiler and allows the creation of some clever tools for generating code.

There is a secondary benefit as well. Developers have historically wasted extraordinary amounts of time on format wars. Since Go defines a standard way of formatting code, Go developers avoid arguments over Brace Style and Tabs vs. Spaces. For example, Go programs use tabs to indent, and it is a syntax error if the opening brace is not on the same line as the declaration or command that begins the block.

> **NOTE**
>
> Many Go developers think the Go team defined a standard format as a way to avoid developer arguments and discovered the tooling advantages later. However, Russ Cox, the development lead for Go, has publicly stated that better tooling was his original motivation.

The Go development tools include a command, `go fmt`, which automatically fixes the whitespace in your code to match the standard format. However, it can't fix braces on the wrong line. Run it with:

```
$ go fmt ./...
hello.go
```

Using `./...` tells a Go tool to apply the command to all the files in the current directory and all subdirectories. We will see it again as we learn about more Go tools.

If you open up `hello.go` you'll see that the line with `fmt.Println` is now properly indented with a single tab.

TIP

Remember to run `go fmt` before you compile your code, and, at the very least, before you commit source code changes to your repository! If you forget, make a separate commit that *only* does **`go fmt ./...`** so you don't hide logic changes in an avalanche of formatting changes.

THE SEMICOLON INSERTION RULE

The `go fmt` command won't fix braces on the wrong line because of the *semicolon insertion rule*. Like C or Java, Go requires a semicolon at the end of every statement. However, Go developers should never put the semicolons in themselves. The Go compiler adds them automatically following a very simple rule described in Effective Go:

If the last token before a newline is any of the following, the lexer inserts a semicolon after the token:

- An identifier (which includes words like int and float64)
- A basic literal such as a number or string constant
- One of the tokens: "break," "continue," "fallthrough," "return," "++," "--," ")," or "}"

With this simple rule in place, you can see why putting a brace in the wrong place breaks. If you write your code like this:

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

the semicolon insertion rule sees the ")" at the end of the `func main()` line and turns that into:

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

and that's not valid Go.

The semicolon insertion rule and the resulting restriction on brace placement is one of the things that makes the Go compiler simpler and faster, while at the same time enforcing a coding style. That's clever.

go vet

There is a class of bugs where the code is syntactically valid, but quite likely incorrect. The `go` tool includes a command called `go vet` to detect these kinds of errors. Let's add one to our program and watch it get detected. Change the `fmt.Println` line in `hello.go` to the following:

```
        fmt.Printf("Hello, %s!\n")
```

Example 1-1.

`fmt.Printf` is very similar to `printf` in C, Java, Ruby, and many other languages. If you haven't seen `fmt.Printf` before, it is a function with a template for its first parameter, and values for the placeholders in the template in the remaining parameters.

In this example, we have a template (`"Hello, %s!\n"`) with a `%s` placeholder, but no value was specified for the placeholder. This code will compile and run, but it's not correct. One of the things that `go vet` can detect is whether or not there is a value for every placeholder in a formatting template. Run `go vet` on our modified code and it finds an error:

```
$ go vet ./...
# hello_world
./hello.go:6:2: fmt.Printf format %s reads arg #1, but call has 0 args
```

Now that `go vet` found our bug, it's easy for us to fix it. Change line 6 in `hello.go` to:

```
        fmt.Printf("Hello, %s!\n", "world")
```

While `go vet` catches several common programming errors, there are things that it cannot detect. Luckily, there are third-party Go code-quality

tools to close the gap. Some of the most popular code quality tools are covered in "Code-quality scanners".

> **TIP**
>
> Just like you should run `go fmt` to make sure your code is formatted properly, run `go vet` to scan for possible bugs in valid code. They are just the first step in ensuring that your code is of high quality. In addition to the advice in this book, all Go developers should read through Effective Go and the Code Review Comments page on Go's wiki to understand what idiomatic Go code looks like.

Choose Your Tools

While we wrote a small Go program using nothing more than a text editor and the `go` command, you'll probably want more advanced tools when working on larger projects. Go IDEs provide many advantages over text editors, including automatic formatting on save, code completion, type checking, error reporting, and integrated debugging. There are excellent Go development tools for most text editors and IDEs. If you don't already have a favorite tool, two of the most popular Go development environments are Visual Studio Code and GoLand.

Visual Studio Code

If you are looking for a free development environment, Visual Studio Code from Microsoft is your best option. Since it was released in 2015, VS Code has become the most popular source code editor for developers. It does not ship with Go support, but you can make it a Go development environment by downloading the Go extension from the extensions gallery.

VS Code's Go support relies on third-party extensions that are accessed via its built-in Marketplace. This includes the Go Development tools, The Delve debugger, and gopls, a Go Language Server developed by the Go team. While you need to install the Go compiler yourself, the Go extension will install Delve and gopls for you.

> **NOTE**
>
> What is a language server? It's a standard specification for an API that enables editors to implement intelligent editing behavior, like code completion, quality checks, or finding all the places a variable or function is used in your code. You can learn more about language servers and their capabilities by checking out the language server protocol website.

Once your tools are set up, you can open your project and work with it. Figure 1-1 shows you what your project window should look like. Getting Started with VS Code Go is a walkthrough that demonstrates the VS Code Go extension.

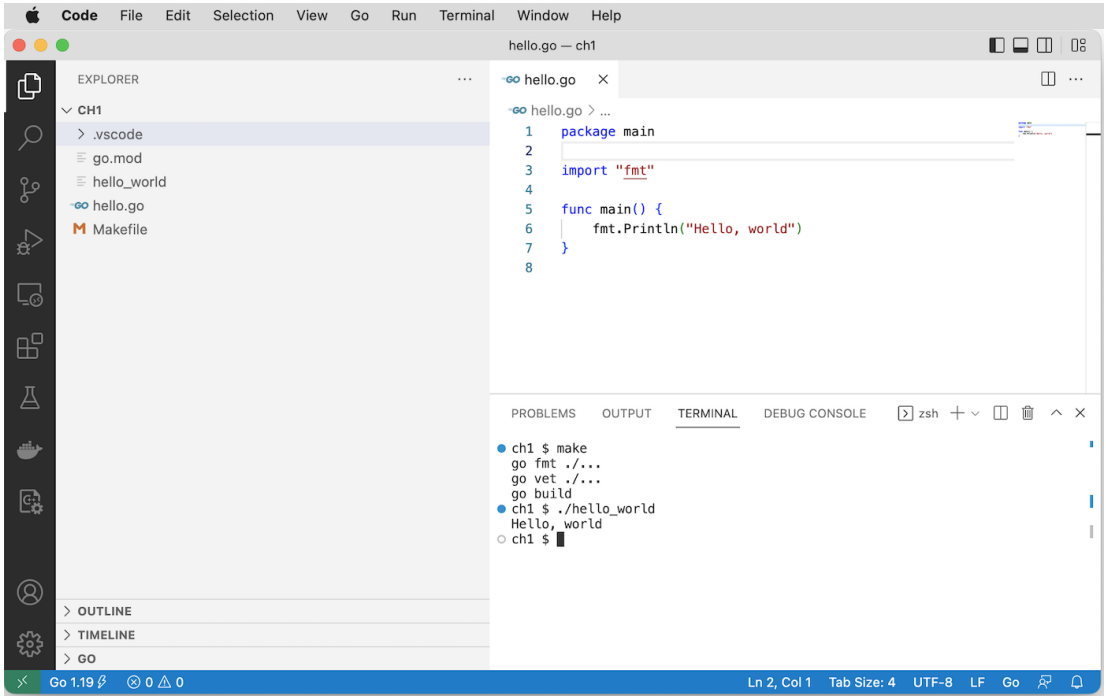

Figure 1-1. Visual Studio Code

GoLand

GoLand is the Go-specific IDE from JetBrains. While JetBrains is best known for Java-centric tools, GoLand is an excellent Go development environment. As you can see in Figure 1-2, GoLand's user interface looks similar to IntelliJ, PyCharm, RubyMine, WebStorm, Android Studio, or any

of the other JetBrains IDEs. Its Go support includes refactoring, syntax highlighting, code completion and navigation, documentation pop-ups, a debugger, code coverage, and more. In addition to Go support, GoLand includes JavaScript/HTML/CSS and SQL database tools. Unlike VS Code, GoLand doesn't require you to install a plugin to get it to work.

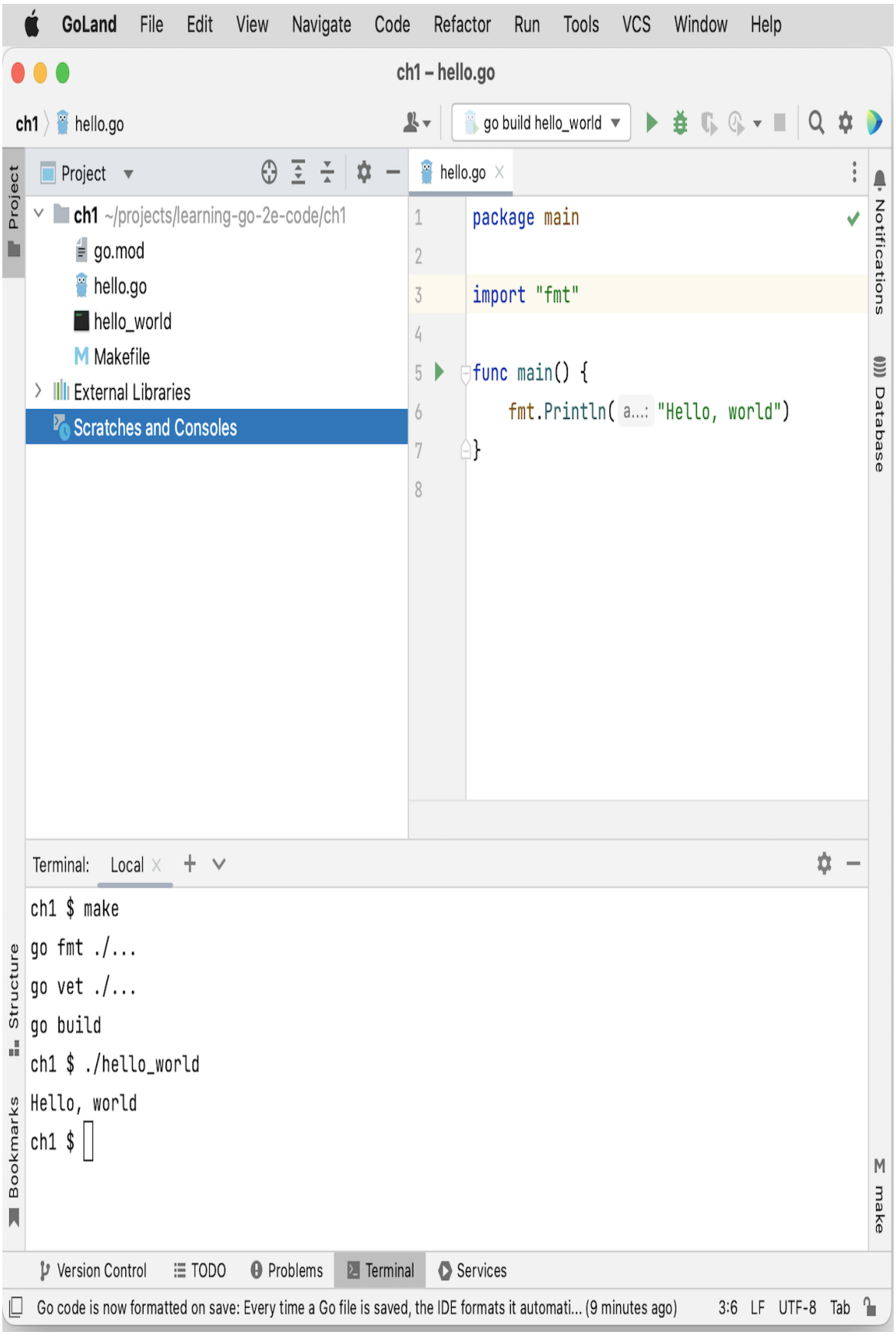
GoLand File Edit View Navigate Code Refactor Run Tools VCS Window Help
ch1 – hello.go
ch1 hello.go
go build hello_world
Project
ch1 ~/projects/learning-go-2e-code/ch1
go.mod
hello.go
hello_world
Makefile
External Libraries
Scratches and Consoles
hello.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println(a...: "Hello, world")
7 }
8
Terminal: Local
ch1 $ make
go fmt ./...
go vet ./...
go build
ch1 $ ./hello_world
Hello, world
ch1 $
Notifications
Database
Structure
Bookmarks
make
Version Control
TODO
Problems
Terminal
Services
Go code is now formatted on save: Every time a Go file is saved, the IDE formats it automati... (9 minutes ago)
3:6 LF UTF-8 Tab

Figure 1-2. GoLand

If you have already subscribed to IntelliJ Ultimate, you can add Go support via a plug-in. While GoLand is commercial software, Jetbrains has a Free License Program for students and core open-source contributors. If you don't qualify for a free license, there is a 30-day free trial available. After that, you have to pay for GoLand.

The Go Playground

There's one more important tool for Go development, but this is one that you don't install. Visit The Go Playground and you'll see a window that resembles Figure 1-3. If you have used a command-line environment like `irb`, `node`, or `python`, you'll find The Go Playground has a very similar feel. It gives you a place to try out and share small programs. Enter your program into the window and click the Run button to execute the code. The Format button runs `go fmt` on your program and updates your imports. The Share button creates a unique URL that you can send to someone else to take a look at your program or to come back to your code at a future date (the URLs have proven to be persistent for a long time, but I wouldn't rely on the playground as your source code repository).

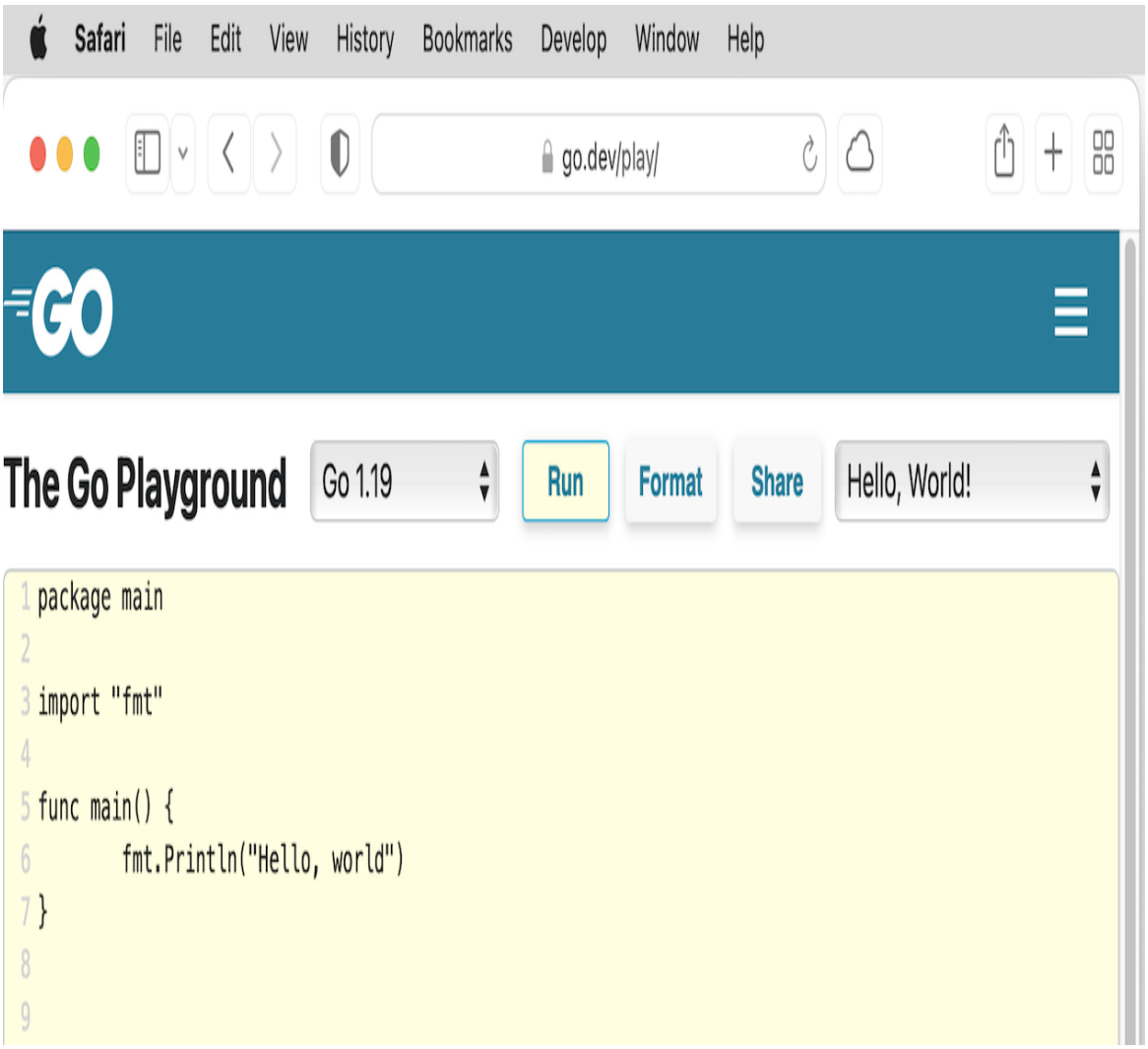

Figure 1-3. The Go Playground

As you can see in Figure 1-4, you can simulate multiple files by separating each file with a line that looks like `-- filename.go --`. You can even create simulated subdirectories by including a `/` in the filename, such as `-- subdir/my_code.go --` .

Be aware that The Go Playground is on someone else's computer (in particular, Google's computer), so you don't have completely free rein. It gives you a choice of a few versions of Go (usually the current release, the previous release, and the latest development version). You can only make network connections to `localhost`, and processes that run for too long or use too much memory are stopped. If your program depends on time, you need to take into account that the clock is set to November 10, 2009,

23:00:00 UTC (the date of the initial announcement of Go). Even with these limitations, The Go Playground is a very useful way to try out new ideas without creating a new project locally. Throughout this book, you'll find links to The Go Playground so you can run code examples without copying them onto your computer.

WARNING

Do not put sensitive information (such as personally identifiable information, passwords, or private keys) into your playground! If you click the Share button, the information is saved on Google's servers and is accessible to anyone who has the associated Share URL. If you do this by accident, contact Google at security@golang.org with the URL and the reason the content needs to be removed.

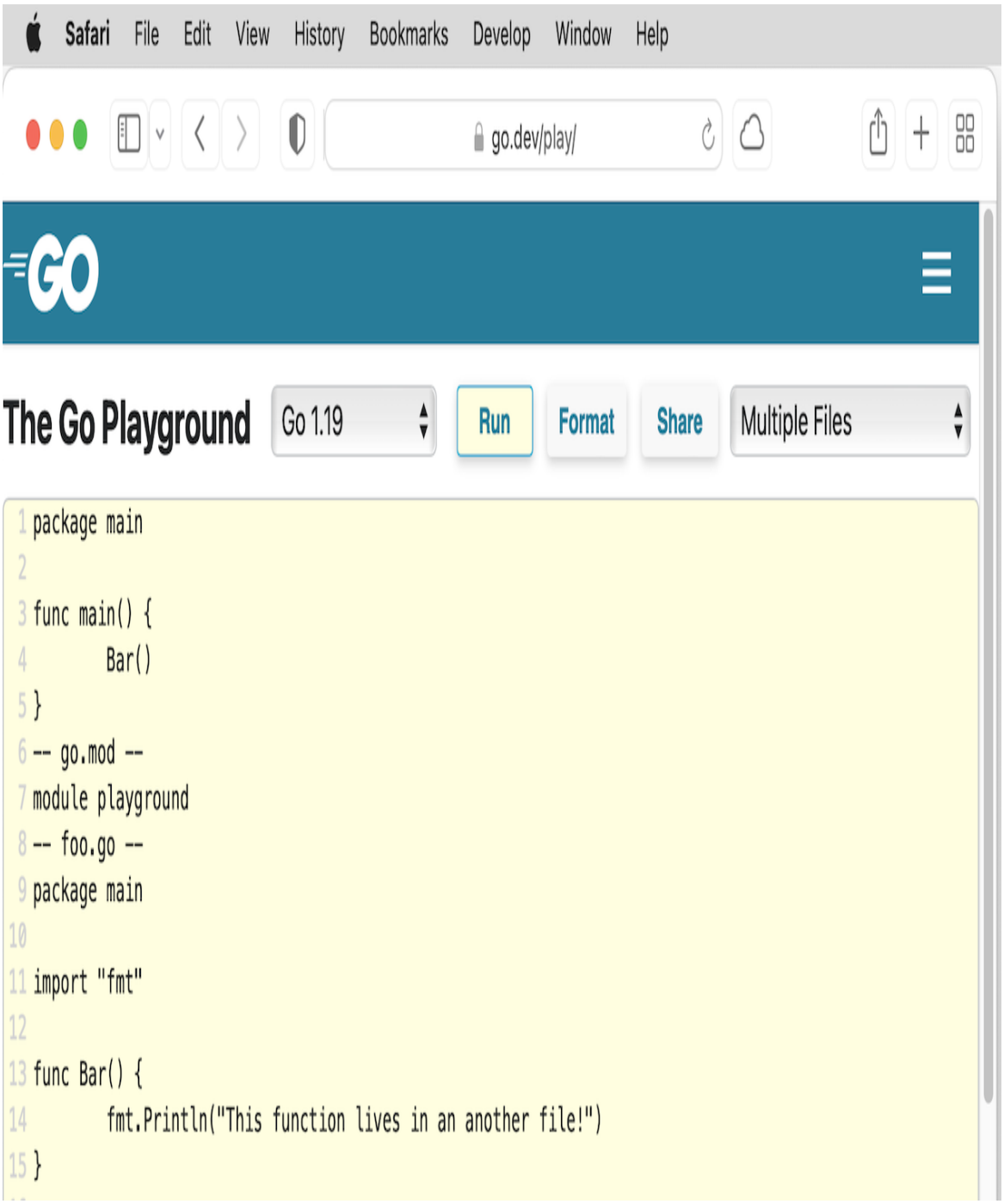

Figure 1-4. The Go Playground supports multiple files

Makefiles

An IDE is nice to use, but it's hard to automate. Modern software development relies on repeatable, automatable builds that can be run by

anyone, anywhere, at any time. Requiring this kind of tooling is good software engineering practice. It avoids the age-old situation, where a developer absolves themselves of any build problems with a shrug and the statement, "It works on my machine!" The way to do this is to use some kind of script to specify your build steps. Go developers have adopted `make` as their solution. It lets developers specify a set of operations that are necessary to build a program and the order in which the steps must be performed. You may not be familiar with `make`, but it's been used to build programs on Unix systems since 1976.

Create a file called `Makefile` in the *ch1* directory with the following contents:

```
.DEFAULT_GOAL := build

.PHONY:fmt vet build
fmt:
	go fmt ./...

vet: fmt
	go vet ./...

build: vet
	go build
```

Even if you haven't seen a Makefile before, it's not too difficult to figure out what is going on. Each possible operation is called a *target*. The `.DEFAULT_GOAL` defines which target is run when no target is specified. In our case, we are going to run the `build` target. Next we have the target definitions. The word before the colon (`:`) is the name of the target. Any words after the target (like `vet` in the line `build: vet`) are the other targets that must be run before the specified target runs. The tasks that are performed by the target are on the indented lines after the target. The `.PHONY` line keeps `make` from getting confused if there is a directory or file in your project with same name as one of the listed targets.

Run `make` and you should see the following output:

```
$ make
go fmt ./...
go vet ./...
go build
```

By entering a single command, we format the code correctly, check it for nonobvious errors, and compile it. We can also vet the code with `make vet`, or just run the formatter with `make fmt`. This might not seem like a big improvement, but ensuring that formatting and vetting always happen before a developer (or a script running on a continuous integration build server) triggers a build means you won't miss any steps.

One drawback to Makefiles is that they are exceedingly picky. You *must* indent the steps in a target with a tab. They are also not supported out-of-the-box on Windows. If you are doing your Go development on a Windows computer, you need to install `make` first. The easiest way to do so is to first install a package manager like Chocolatey and then use it to install `make` (for Chocolatey, the command is `choco install make`.)

If you want to learn more about writing Makefiles, there's a good tutorial by Chase Lambert, but it does use a tiny bit of C to explain the concepts.

You can find the code from this chapter in the ch1 repository for Learning Go 2nd Edition on Github.

The Go Compatibility Promise

As with all programming languages, there are periodic updates to the Go development tools. Since Go 1.2, there has been a new release roughly every six months. There are also patch releases with bug and security fixes released as needed. Given the rapid development cycles and the Go team's commitment to backward compatibility, Go releases tend to be incremental rather than expansive. The Go Compatibility Promise is a detailed description of how the Go team plans to avoid breaking Go code. It says that there won't be backward-breaking changes to the language or the standard library for any Go version that starts with 1, unless the change is required for a bug or security fix. In his GopherCon 2022 keynote talk

Compatibility: How Go Programs Keep Working, Russ Cox discusses all the ways that the Go Team works to keep Go code from breaking. He says, "I believe that prioritizing compatibility was the most important design decision that we made in Go 1."

This guarantee doesn't apply to the `go` commands. There have been backward-incompatible changes to the flags and functionality of the `go` commands, and it's entirely possible that it will happen again.

Staying Up to Date

Go programs compile to a standalone native binary, so you don't need to worry that updating your development environment could cause your currently deployed programs to fail. You can have programs compiled with different versions of Go running simultaneously on the same computer or virtual machine.

When you are ready to update the Go development tools installed on your computer, Mac and Windows users have the easiest path. Those who installed with `brew` or `chocolatey` can use those tools to update. Those who used the installers on *https://golang.org/dl* can download the latest installer, which removes the old version when it installs the new one.

Linux and BSD users need to download the latest version, move the old version to a backup directory, unpack the new version, and then delete the old version:

```
$ mv /usr/local/go /usr/local/old-go
$ tar -C /usr/local -xzf go1.19.6.linux-amd64.tar.gz
$ rm -rf /usr/local/old-go
```

> **NOTE**
>
> Technically, you don't need to move the existing installation to a new location; you could just delete it and install the new version. However, this falls in the "better safe than sorry" category. If something goes wrong while installing the new version, it's good to have the previous one around.

Exercises

> **NOTE**
>
> Each chapter will have some exercises at the end to let you try out the ideas that we cover. Answers to the exercises are found in the ch1 repository for Learning Go 2nd Edition on Github .

1. Take our "Hello, world!" program and run it on the Go Playground. Share a link to the code in the playground with a co-worker who would love to learn about Go.
2. Add a target to the Makefile called `clean` that removes the `hello_world` binary and any other temporary files created by `go build`. Take a look at the Go command documentation to find a `go` command to help implement this.
3. Experiment with modifying the formatting in the "Hello, world!" program. Add blank lines, spaces, change indentation, insert newlines. After making a modification, run `go fmt` to see if the formatting change is undone. Also, run `go build` to see if the code still compiles. You can also add additional `fmt.Println` calls so you can see what happens if you put blank lines in the middle of a function.

Wrapping Up

In this chapter, we learned how to install and configure our Go development environment. We also talked about tools for building Go programs and ensuring code quality. Now that our environment is ready, we're on to our next chapter, where we explore the built-in types in Go and how to declare variables.

Chapter 2. Predeclared Types and Declarations

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Now that we have our development environment set up, it's time to start looking at Go's language features and how to best use them. When trying to figure out what "best" means, there is one overriding principle: write your programs in a way that makes your intentions clear. As we go through features, we'll look at the options and I'll explain why I find a particular approach produces clearer code.

We'll start by looking at the types that are built into Go and how to declare variables of those types. While every programmer has experience with these concepts, Go does some things differently, and there are subtle differences between Go and other languages.

The Predeclared Types

Go has many types built into the languages. These are called *predeclared* types. They are similar to types that are found in other languages: booleans, integers, floats, and strings. Using these types idiomatically is sometimes a challenge for developers who are transitioning from another language. We are going to look at these types and see how they work best in Go. Before we review the types, let's cover some of the concepts that apply to all types.

The Zero Value

Go, like most modern languages, assigns a default *zero value* to any variable that is declared but not assigned a value. Having an explicit zero value makes code clearer and removes a source of bugs found in C and C++ programs. As we talk about each type, we will also cover the zero value for the type.

Literals

A Go *literal* is an explicitly specified number, character, or string. There are four common kinds of literals in Go programs. (There's a rare fifth kind of literal that we'll cover when discussing complex numbers.)

An *integer literal* is a sequence of numbers. Integer literals are base ten by default, but different prefixes are used to indicate other bases: 0b for binary (base two), 0o for octal (base eight), or 0x for hexadecimal (base sixteen). You can use either or upper- or lowercase letters for the prefix. A leading 0 with no letter after it is another way to represent an octal literal. Do not use it, as it is very confusing.

To make it easier to read longer integer literals, Go allows you to put underscores in the middle of your literal. This allows you to, for example, group by thousands in base ten (1_234). These underscores have no effect on the value of the number. The only limitations on underscores are that they can't be at the beginning or end of numbers, and you can't have them next to each other. You could put an underscore between every digit in your literal (1_2_3_4), but don't. Use them to improve readability by breaking

up base ten numbers at the thousands place or to break up binary, octal, or hexadecimal numbers at one-, two-, or four-byte boundaries.

A *floating point literal* has a decimal point to indicate the fractional portion of the value. They can also have an exponent specified with the letter e and a positive or negative number (such as 6.03e23). You also have the option to write them in hexadecimal by using the 0x prefix and the letter `p` for indicating any exponent (0x12.34p5, which is equal to 582.5 in base 10). Like integer literals, you can use underscores to format your floating point literals.

A *rune literal* represents a character and is surrounded by single quotes. Unlike many other languages, in Go single quotes and double quotes are *not* interchangeable. Rune literals can be written as single Unicode characters (`'a'`), 8-bit octal numbers (`'\141'`), 8-bit hexadecimal numbers (`'\x61'`), 16-bit hexadecimal numbers (`'\u0061'`), or 32-bit Unicode numbers (`'\U00000061'`). There are also several backslash escaped rune literals, with the most useful ones being newline (`'\n'`), tab (`'\t'`), single quote (`'\''`), and backslash (`'\\'`).

Practically speaking, use base ten to represent your integer and floating point literals. Octal representations are rare, mostly used to represent POSIX permission flag values (such as 0o777 for rwxrwxrwx). Hexadecimal and binary are sometimes used for bit filters or networking and infrastructure applications. Avoid using any of the numeric escapes for rune literals, unless the context makes your code clearer.

There are two different ways to indicate *string literals*. Most of the time, you should use double quotes to create an *interpreted string literal* (e.g., type `"Greetings and Salutations"`). These contain zero or more rune literals. They are called “interpreted” because they interpret rune literals (both numeric and backslash escaped) into single characters.

> **NOTE**
>
> There is one rune literal backslash escape that's not legal in a string literal: the single quote escape. It is replaced by a backslash escape for double quotes.

The only characters that cannot appear in an interpreted string literal are unescaped backslashes, unescaped newlines, and unescaped double quotes. If you use an interpreted string literal and want your greetings on a different line from your salutations and you want "Salutations" to appear in quotes, you need to type **`"Greetings and\n\"Salutations\""`**.

If you need to include backslashes, double quotes, or newlines in your string, it's easier to use a *raw string literal*. These are delimited with backquotes (`) and can contain any character except a backquote. There's no escape character in a raw string literal; all characters are included as-is. When using a raw string literal, we write our multiline greeting like so:

```
`Greetings and
"Salutations"`
```

Literals are considered *untyped*. We'll explore this concept more in "Literals are Untyped". As you will see in "var Versus :=", there are situations in Go where the type isn't explicitly declared. In those cases, Go uses the *default type* for a literal; if there's nothing in the expression that makes clear what the type of the literal is, the literal defaults to a type. We will mention the default type for literals as we look at the different predeclared types.

Booleans

The `bool` type represents Boolean variables. Variables of `bool` type can have one of two values: `true` or `false`. The zero value for a `bool` is `false`:

```
var flag bool // no value assigned, set to false
var isAwesome = true
```

It's hard to talk about variable types without showing a variable declaration, and vice versa. We'll use variable declarations first and describe them in "var Versus :=".

Numeric Types

Go has a large number of numeric types: 12 different types (and a few special names) that are grouped into three categories. If you are coming from a language like JavaScript that gets along with only a single numeric type, this might seem like a lot. And in fact, some types are used frequently while others are more esoteric. We'll start by looking at integer types before moving on to floating point types and the very unusual complex type.

Integer types

Go provides both signed and unsigned integers in a variety of sizes, from one to eight bytes. They are shown in Table 2-1.

Table 2-1. The integer types in Go

Type name	Value range
int8	–128 to 127
int16	–32768 to 32767
int32	–2147483648 to 2147483647
int64	–9223372036854775808 to 9223372036854775807
uint8	0 to 255
uint16	0 to 65535
uint32	0 to 4294967295
uint64	0 to 18446744073709551615

It might be obvious from the name, but the zero value for all of the integer types is `0`.

The special integer types

Go does have some special names for integer types. A `byte` is an alias for `uint8`; it is legal to assign, compare, or perform mathematical operations between a `byte` and a `uint8`. However, you rarely see `uint8` used in Go code; just call it a `byte`.

The second special name is `int`. On a 32-bit CPU, `int` is a 32-bit signed integer like an `int32`. On most 64-bit CPUs, `int` is a 64-bit signed integer, just like an `int64`. Because `int` isn't consistent from platform to platform,

it is a compile-time error to assign, compare, or perform mathematical operations between an `int` and an `int32` or `int64` without a type conversion (see "Explicit Type Conversion" for more details). Integer literals default to being of `int` type.

> **NOTE**
>
> There are some uncommon 64-bit CPU architectures that use a 32-bit signed integer for the `int` type. Go supports three of them: amd64p32, mips64p32, and mips64p32le.

The third special name is `uint`. It follows the same rules as `int`, only it is unsigned (the values are always 0 or positive).

There are two other special names for integer types, `rune` and `uintptr`. We looked at rune literals earlier and discuss the `rune` type in "A Taste of Strings and Runes" and `uintptr` in Chapter 16.

Choosing which integer to use

Go provides more integer types than some other languages. Given all of these choices, you might wonder when you should use each of them. There are three simple rules to follow:

- If you are working with a binary file format or network protocol that has an integer of a specific size or sign, use the corresponding integer type.
- If you are writing a library function that should work with any integer type, take advantage of Go's generics support and use a generic type parameter to represent any integer type (We talk more about functions and their parameters in Chapter 5 and more about generics in Chapter 8.)
- In all other cases, just use `int`.

> **NOTE**
>
> It's likely that you'll find legacy code where there's a pair of functions that do the same thing, but one has `int64` for the parameters and variables and the other has `uint64`. The reason why is that the API was created before generics were added to Go. Without generics, you needed to write functions with slightly different names to implement the same algorithm with different types. Using `int64` and `uint64` meant that you could write the code once and let callers use type conversions to pass values in and convert data that's returned.
>
> You can see this pattern in the Go standard library within the functions `FormatInt` and `FormatUint` in the `strconv` package.

Integer operators

Go integers support the usual arithmetic operators: +, `-`, `*`, `/`, with `%` for modulus. The result of an integer division is an integer; if you want to get a floating point result, you need to use a type conversion to make your integers into floating point numbers. Also, be careful not to divide an integer by 0; this causes a panic (we talk more about panics in "panic and recover").

> **NOTE**
>
> Integer division in Go follows truncation toward zero; see the Go spec's section on arithmetic operators for the full details.

You can combine any of the arithmetic operators with = to modify a variable: `+=`, `-=`, `*=`, `/=`, and `%=`. For example, the following code results in `x` having the value 20:

```
var x int = 10
x *= 2
```

You compare integers with `==`, `!=`, `>`, `>=`, `<`, and `<=`.

Go also has bit-manipulation operators for integers. You can bit shift left and right with `<<` and `>>`, or do bit masks with `&` (logical AND), `|` (logical

OR), `^` (logical XOR), and `&^` (logical AND NOT). Just like the arithmetic operators, you can also combine all of the logical operators with `=` to modify a variable: `&=`, `|=`, `^=`, `&^=`, `<<=`, and `>>=`.

Floating point types

There are two floating point types in Go, as shown in Table 2-2.

Table 2-2. The floating point types in Go

Type name	Largest absolute value
float32	3.40282346638528859811704183484516925440e+38
float64	1.797693134862315708145274237317043567981e+308

Like the integer types, the zero value for the floating point types is 0.

Floating point in Go is similar to floating point math in other languages. Go uses the IEEE 754 specification, giving a large range and limited precision. Picking which floating point type to use is straightforward: unless you have to be compatible with an existing format, use `float64`. Floating point literals have a default type of `float64`, so always using `float64` is the simplest option. It also helps mitigate floating point accuracy issues since a `float32` only has six- or seven-decimal digits of precision. Don't worry about the difference in memory size unless you have used the profiler to determine that it is a significant source of problems. (Testing and profiling are covered in Chapter 15.)

The bigger question is whether you should be using a floating point number at all. In most cases, the answer is no. Just like other languages, Go floating point numbers have a huge range, but they cannot store every value in that range; they store the nearest approximation. Because floats aren't exact,

they can only be used in situations where inexact values are acceptable or the rules of floating point are well understood. That limits them to things like graphics and scientific operations.

> **WARNING**
>
> A floating point number cannot represent a decimal value exactly. Do not use them to represent money or any other value that must have an exact decimal representation! We'll look at a third-party module for handling exact decimal values in "Importing Third-Party Code".

IEEE 754

As mentioned earlier, Go (and most other programming languages) stores floating point numbers using a specification called IEEE 754.

The actual rules are outside the scope of this book, and they aren't straightforward. You can learn more about IEEE 754 from The Floating Point Guide.

You can use all the standard mathematical and comparison operators with floats, except `%`. Floating point division has a couple of interesting properties. Dividing a nonzero floating point variable by 0 returns `+Inf` or `-Inf` (positive or negative infinity), depending on the sign of the number. Dividing a floating point variable set to 0 by 0 returns `NaN` (Not a Number).

While Go lets you use `==` and `!=` to compare floats, don't do it. Due to the inexact nature of floats, two floating point values might not be equal when you think they should be. Instead, define a maximum allowed variance and see if the difference between two floats is less than that. This value (sometimes called *epsilon*) depends on what your accuracy needs are; I can't give you a simple rule. If you aren't sure, consult your friendly local mathematician for advice. If you can't find one, The Floating Point Guide has another page that can help you out (or possibly convince you to avoid floating point numbers unless absolutely necessary).

Complex types (you're probably not going to use these)

There is one more numeric type and it is pretty unusual. Go has first-class support for complex numbers. If you don't know what complex numbers are, you are not the target audience for this feature; feel free to skip ahead.

There isn't a lot to the complex number support in Go. Go defines two complex number types. `complex64` uses `float32` values to represent the real and imaginary part, and `complex128` uses `float64` values. Both are declared with the `complex` built-in function. Go uses a few rules to determine what the type of the function output is:

- If you use untyped constants or literals for both function parameters, you'll create an untyped complex literal, which has a default type of `complex128`.
- If both of the values passed into `complex` are of `float32` type, you'll create a `complex64`.
- If one value is a `float32` and the other value is an untyped constant or literal that can fit within a `float32`, you'll create a `complex64`.
- Otherwise, you'll create a `complex128`.

All of the standard floating point arithmetic operators work on complex numbers. Just like floats, you can use `==` or `!=` to compare them, but they have the same precision limitations, so it's best to use the epsilon technique. You can extract the real and imaginary portions of a complex number with the `real` and `imag` built-in functions, respectively. There are also some additional functions in the `math/cmplx` package for manipulating `complex128` values.

The zero value for both types of complex numbers has 0 assigned to both the real and imaginary portions of the number.

Example 2-1 shows a simple program that demonstrates how complex numbers work. You can run it for yourself on The Go Playground.

Example 2-1. Complex numbers

```
func main() {
    x := complex(2.5, 3.1)
    y := complex(10.2, 2)
    fmt.Println(x + y)
    fmt.Println(x - y)
    fmt.Println(x * y)
    fmt.Println(x / y)
    fmt.Println(real(x))
    fmt.Println(imag(x))
    fmt.Println(cmplx.Abs(x))
}
```

Running this code gives you:

```
(12.7+5.1i)
(-7.699999999999999+1.1i)
(19.3+36.62i)
(0.2934098482043688+0.24639022584228065i)
2.5
3.1
3.982461550347975
```

You can see floating point imprecision on display here, too.

In case you were wondering what the fifth kind of primitive literal was, Go supports imaginary literals to represent the imaginary portion of a complex number. They look just like floating point literals, but they have an `i` for a suffix.

Despite having complex numbers as a predeclared type, Go is not a popular language for numerical computing. Adoption has been limited because other features (like matrix support) are not part of the language and libraries have to use inefficient replacements, like slices of slices. (We'll look at slices in Chapter 3 and how they are implemented in Chapter 6.) But if you need to calculate a Mandelbrot set as part of a larger program, or implement a quadratic equation solver, complex number support is there for you.

You might be wondering why Go includes complex numbers. The answer is simple: Ken Thompson, one of the creators of Go (and Unix), thought they would be interesting. There has been discussion about removing complex

numbers from a future version of Go, but it's easier to just ignore the feature.

> **NOTE**
>
> If you do want to write numerical computing applications in Go, you can use the third-party Gonum package. It takes advantage of complex numbers and provides useful libraries for things like linear algebra, matrices, integration, and statistics. But you should consider other languages first.

A Taste of Strings and Runes

This brings us to strings. Like most modern languages, Go includes strings as a built-in type. The zero value for a string is the empty string. Go supports Unicode; as we showed in the section on string literals, you can put any Unicode character into a string. Like integers and floats, strings are compared for equality using `==`, difference with `!=`, or ordering with `>`, `>=`, `<`, or `<=`. They are concatenated by using the `+` operator.

Strings in Go are immutable; you can reassign the value of a string variable, but you cannot change the value of the string that is assigned to it.

Go also has a type that represents a single code point. The *rune* type is an alias for the `int32` type, just like `byte` is an alias for `uint8`. As you could probably guess, a rune literal's default type is a rune, and a string literal's default type is a string.

If you are referring to a character, use the rune type, not the `int32` type. They might be the same to the compiler, but you want to use the type that clarifies the intent of your code.

```
var myFirstInitial rune = 'J' // good - the type name matches the usage
var myLastInitial int32 = 'B' // bad - legal but confusing
```

We are going to talk a lot more about strings in the next chapter, covering some implementation details, relationship with bytes and runes, as well as advanced features and pitfalls.

Explicit Type Conversion

Most languages that have multiple numeric types automatically convert from one to another when needed. This is called *automatic type promotion*, and while it seems very convenient, it turns out that the rules to properly convert one type to another can get complicated and produce unexpected results. As a language that values clarity of intent and readability, Go doesn't allow automatic type promotion between variables. You must use a *type conversion* when variable types do not match. Even different-sized integers and floats must be converted to the same type to interact. This makes it clear exactly what type you want without having to memorize any type conversion rules (see Example 2-2).

Example 2-2. Type conversions

```
var x int = 10
var y float64 = 30.2
var sum1 float64 = float64(x) + y
var sum2 int = x + int(y)
fmt.Println(sum1, sum2)
```

In this sample code we define four variables. `x` is an `int` with the value 10, and `y` is a `float64` with the value 30.2. Since these are not identical types, we need to convert them to add them together. For `sum1`, we convert `x` to a `float64` using a `float64` type conversion, and for `sum2`, we convert `y` to an `int` using an `int` type conversion. When you run this code, it prints out 40.2 40.

The same behavior applies with different-sized integer types (see Example 2-3).

Example 2-3. Integer Type Conversions

```
var x int = 10
var b byte = 100
var sum3 int = x + int(b)
var sum4 byte = byte(x) + b
fmt.Println(sum3, sum4)
```

You can run these examples on The Go Playground.

This strictness around types has other implications. Since all type conversions in Go are explicit, you cannot treat another Go type as a boolean. In many languages, a nonzero number or a nonempty string can be interpreted as a boolean `true`. Just like automatic type promotion, the rules for "truthy" values vary from language to language and can be confusing. Unsurprisingly, Go doesn't allow truthiness. In fact, *no other type can be converted to a bool, implicitly or explicitly*. If you want to convert from another data type to boolean, you must use one of the comparison operators (`==`, `!=`, `>`, `<`, `<=`, or `>=`). For example, to check if variable `x` is equal to 0, the code would be `x == 0`. If you want to check if string `s` is empty, use `s == ""`.

> **NOTE**
>
> Type conversions are one of the places where Go chooses to add a little verbosity in exchange for a great deal of simplicity and clarity. You'll see this trade-off multiple times. Idiomatic Go values comprehensibility over conciseness.

Literals are Untyped

While you can't add two integer variables together if they are declared to be of different types of integers, Go lets you use an integer literal in floating point expressions or even assign an integer literal to a floating point variable.

```
var x float64 = 10
var y float64 = 200.3 * 5
```

This is because, as we mentioned earlier, literals in Go are untyped. Go is a practical language and it makes sense to avoid forcing a type until the developer specifies one. This means they can be used with any variable whose type is compatible with the literal. When we look at user-defined types in Chapter 7, you'll see that we can even use literals with user-defined types based on predefined types. Being untyped only goes so far; you can't assign a literal string to a variable with a numeric type or a literal number to

a string variable, nor can you assign a float literal to an int. These are all flagged by the compiler as errors. There are also size limitations; while you can write numeric literals that are larger than any integer can hold, it is a compile-time error to try to assign a literal whose value overflows the specified variable, such as trying to assign the literal 1000 to a variable of type `byte`.

var Versus :=

For a small language, Go has a lot of ways to declare variables. There's a reason for this: each declaration style communicates something about how the variable is used. Let's go through the ways you can declare a variable in Go and see when each is appropriate.

The most verbose way to declare a variable in Go uses the `var` keyword, an explicit type, and an assignment. It looks like this:

```
var x int = 10
```

If the type on the righthand side of the `=` is the expected type of your variable, you can leave off the type from the left side of the `=`. Since the default type of an integer literal is `int`, the following declares `x` to be a variable of type `int`:

```
var x = 10
```

Conversely, if you want to declare a variable and assign it the zero value, you can keep the type and drop the `=` on the righthand side:

```
var x int
```

You can declare multiple variables at once with `var`, and they can be of the same type:

```
var x, y int = 10, 20
```

all zero values of the same type:

```
var x, y int
```

or of different types:

```
var x, y = 10, "hello"
```

There's one more way to use `var`. If you are declaring multiple variables at once, you can wrap them in a *declaration list*:

```
var (
    x    int
    y        = 20
    z    int = 30
    d, e     = 40, "hello"
    f, g string
)
```

Go also supports a short declaration and assignment format. When you are within a function, you can use the `:=` operator to replace a `var` declaration that uses type inference. The following two statements do exactly the same thing: they declare `x` to be an `int` with the value of 10:

```
var x = 10
x := 10
```

Like `var`, you can declare multiple variables at once using `:=`. These two lines both assign 10 to `x` and "hello" to `y`:

```
var x, y = 10, "hello"
x, y := 10, "hello"
```

The `:=` operator can do one trick that you cannot do with `var`: it allows you to assign values to existing variables, too. As long as there is at least one new variable on the lefthand side of the `:=`, then any of the other variables can already exist:

```
x := 10
x, y := 30, "hello"
```

There is one limitation on `:=`. If you are declaring a variable at package level, you must use `var` because `:=` is not legal outside of functions.

How do you know which style to use? As always, choose what makes your intent clearest. The most common declaration style within functions is `:=`. Outside of a function, use declaration lists on the rare occasions when you are declaring multiple package-level variables.

There are some situations within functions where you should avoid `:=`:

- When initializing a variable to its zero value, use `var x int`. This makes it clear that the zero value is intended.
- When assigning an untyped constant or a literal to a variable and the default type for the constant or literal isn't the type you want for the variable, use the long `var` form with the type specified. While it is legal to use a type conversion to specify the type of the value and use `:=` to write `x := byte(20)`, it is idiomatic to write `var x byte = 20`.
- Because `:=` allows you to assign to both new and existing variables, it sometimes creates new variables when you think you are reusing existing ones (see "Shadowing" for details). In those situations, explicitly declare all of your new variables with `var` to make it clear which variables are new, and then use the assignment operator (`=`) to assign values to both new and old variables.

While `var` and `:=` allow you to declare multiple variables on the same line, only use this style when assigning multiple values returned from a function or the comma ok idiom (see Chapter 5).

You should rarely declare variables outside of functions, in what's called the *package block* (see "Blocks" in Chapter 4). Package-level variables whose values change are a bad idea. When you have a variable outside of a function, it can be difficult to track the changes made to it, which makes it hard to understand how data is flowing through your program. This can lead

to subtle bugs. As a general rule, you should only declare variables in the package block that are effectively immutable.

> **TIP**
>
> Avoid declaring variables outside of functions because they complicate data flow analysis.

You might be wondering: does Go provide a way to *ensure* that a value is immutable? It does, but it is a bit different from what you may have seen in other programming languages. It's time to learn about `const`.

Using const

Many languages have a way to declare a value as immutable. In Go, this is done with the `const` keyword. At first glance, it seems to work exactly like other languages. Try out the code in Example 2-4 on The Go Playground.

Example 2-4. `const` declarations

```
package main

import "fmt"

const x int64 = 10

const (
    idKey   = "id"
    nameKey = "name"
)

const z = 20 * 10

func main() {
    const y = "hello"

    fmt.Println(x)
    fmt.Println(y)

    x = x + 1 // this will not compile!
    y = "bye" // this will not compile!
```

```
    fmt.Println(x)
    fmt.Println(y)
}
```

When you run this code, compilations fails with the following error messages:

```
./prog.go:20:2: cannot assign to x (constant 10 of type int64)
./prog.go:21:2: cannot assign to y (untyped string constant "hello")
```

As you see, you declare a constant at the package level or within a function. Just like `var`, you can (and should) declare a group of related constants within a set of parentheses.

Be aware that `const` in Go is very limited. Constants in Go are a way to give names to literals. They can only hold values that the compiler can figure out at compile time. This means that they can be assigned:

- Numeric literals
- `true` and `false`
- Strings
- Runes
- The built-in functions `complex`, `real`, `imag`, `len`, and `cap`
- Expressions that consist of operators and the preceding values

> **NOTE**
>
> We'll cover the `len` and `cap` functions in the next chapter. There's another value that can be used with `const` that's called `iota`. We'll talk about `iota` when we discuss creating your own types in Chapter 7.

Go doesn't provide a way to specify that a value calculated at runtime is immutable. For example, the following code will fail to compile with the

error `x + y (value of type int) is not constant`:

```
x := 5
y := 10
const z = x + y // this won't compile!
```

As we'll see in the next chapter, there are no immutable arrays, slices, maps, or structs, and there's no way to declare that a field in a struct is immutable. This is less limiting than it sounds. Within a function, it is clear if a variable is being modified, so immutability is less important. In "Call Value" in Chapter 5, we'll see how Go prevents modifications to variables that are passed as parameters to functions.

> **TIP**
>
> Constants in Go are a way to give names to literals. There is *no* way in Go to declare that a variable is immutable.

Typed and Untyped Constants

Constants can be typed or untyped. An untyped constant works exactly like a literal; it has no type of its own, but does have a default type that is used when no other type can be inferred. A typed constant can only be directly assigned to a variable of that type.

Whether or not to make a constant typed depends on why the constant was declared. If you are giving a name to a mathematical constant that could be used with multiple numeric types, then keep the constant untyped. In general, leaving a constant untyped gives you more flexibility. There are situations where you want a constant to enforce a type. We'll use typed constants when we look at creating enumerations with `iota` in Chapter 7.

Here's what an untyped constant declaration looks like:

```
const x = 10
```

All of the following assignments are legal:

```
var y int = x
var z float64 = x
var d byte = x
```

Here's what a typed constant declaration looks like:

```
const typedX int = 10
```

This constant can only be assigned directly to an `int`. Assigning it to any other type produces a compile-time error like this:

```
cannot use typedX (type int) as type float64 in assignment
```

Unused Variables

One of the goals for Go is to make it easier for large teams to collaborate on programs. To do so, Go has some rules that are unique among programming languages. In Chapter 1, we saw that Go programs need to be formatted in a specific way with `go fmt` to make it easier to write code-manipulation tools and to provide coding standards. Another Go requirement is that *every declared local variable must be read*. It is a *compile-time error* to declare a local variable and to not read its value.

The compiler's unused variable check is not exhaustive. As long as a variable is read once, the compiler won't complain, even if there are writes to the variable that are never read. The following is a valid Go program that you can run on The Go Playground:

```
func main() {
    x := 10 // this assignment isn't read!
    x = 20
    fmt.Println(x)
    x = 30 // this assignment isn't read!
}
```

While the compiler and `go vet` do not catch the unused assignments of 10 and 30 to `x`, there are third-party tools that can detect them. We'll talk about these tools in "Code-quality scanners".

> **NOTE**
>
> The Go compiler won't stop you from creating unread package-level variables. This is one more reason why you should avoid creating package-level variables.

UNUSED CONSTANTS

Perhaps surprisingly, the Go compiler allows you to create unread constants with `const`. This is because constants in Go are calculated at compile time and cannot have any side effects. This makes them easy to eliminate: if a constant isn't used, it is simply not included in the compiled binary.

Naming Variables and Constants

There is a difference between Go's rules for naming variables and the patterns that Go developers follow when naming their variables and constants. Like most languages, Go requires identifier names to start with a letter or underscore, and the name can contain numbers, underscores, and letters. Go's definition of "letter" and "number" is a bit broader than many languages. Any Unicode character that is considered a letter or digit is allowed. This makes all of the variable definitions shown in Example 2-5 perfectly valid Go.

Example 2-5. Variable names you should never use

```
_0 := 0_0
_𝟙 := 20
π := 3
a := "hello" // Unicode U+FF41
__ := "double underscore" // two underscores
```

```
fmt.Println(_0)
fmt.Println(_𝟙)
fmt.Println(ո)
fmt.Println( a )
fmt.Println(__)
```

You can test out this awful code on The Go Playground. While it works, *do not* use variable names like this. These names are considered nonidiomatic because they break the fundamental rule of making sure that your code communicates what it is doing. These names are confusing or difficult to type on many keyboards. Look-alike Unicode code points are the most insidious, because even if they appear to be the same character, they represent entirely different variables. You can run the code shown in Example 2-6 on The Go Playground.

Example 2-6. Using look-alike code points for variable names

```
func main() {
    а := "hello"   // Unicode U+FF41
    a := "goodbye" // standard lowercase a (Unicode U+0061)
    fmt.Println( а )
    fmt.Println(a)
}
```

When you run this program, you get:

```
hello
goodbye
```

Even though underscore is a valid character in a variable name, it is rarely used, because idiomatic Go doesn't use snake case (names like index_counter or number_tries). Instead, idiomatic Go uses camel case (names like indexCounter or numberTries) when an identifier name consists of multiple words.

> **NOTE**
>
> An underscore by itself (_) is a special identifier name in Go; we'll talk more about it when we cover functions in Chapter 5.

In many languages, constants are always written in all uppercase letters, with words separated by underscores (names like INDEX_COUNTER or NUMBER_TRIES). Go does not follow this pattern. This is because Go uses the case of the first letter in the name of a package-level declaration to determine if the item is accessible outside the package. We will revisit this when we talk about packages in Chapter 4.

Within a function, favor short variable names. *The smaller the scope for a variable, the shorter the name that's used for it*. It is very common in Go to see single-letter variable names used with `for` loops. For example, the names `k` and `v` (short for *key* and *value*) are used as the variable names in a `for-range` loop. If you are using a standard `for` loop, `i` and `j` are common names for the index variable. There are other idiomatic ways to name variables of common types; they will be mentioned as we cover more parts of the standard library.

Some languages with weaker type systems encourage developers to include the expected type of the variable in the variable's name. Since Go is strongly typed, you don't need to do this to keep track of the underlying type. However, there are still conventions around variable types and single-letter names. People will use the first letter of a type as the variable name (for example, `i` for integers or `f` for floats). When you define your own types, similar patterns apply.

These short names serve two purposes. The first is that they eliminate repetitive typing, keeping your code shorter. Second, they serve as a check on how complicated your code is. If you find it hard to keep track of your short-named variables, it's likely that your block of code is doing too much.

When naming variables and constants in the package block, use more descriptive names. The type should still be excluded from the name, but since the scope is wider, you need a more complete name to make it clear what the value represents.

Exercises

These exercises demonstrate the concepts discussed in the chapter. Solutions to these exercises, along with the programs in this chapter, are in GitHub at *https://github.com/learning-go-book-2e/ch02* .

1. Write a program where you declare an integer variable called `i` with the value 20. Assign `i` to a floating point variable named `f`. Print out `i` and `f`.
2. Write a program where you declare a constant called `value` that can be assigned to both an integer and a floating point variable. Assign it to an integer called `i` and a floating point variable called `f`. Print out `i` and `f`.
3. Write a program with three variables, one named `b` of type `byte`, one named `smallI` of type `int32`, and one named `bigI` of type `uint64`. Assign each variable the maximum legal value for its type, then add `1` to each variable. Print out their values.

Wrapping Up

We've covered a lot of ground here, understanding how to use the predeclared types, declare variables, and work with assignments and operators. In the next chapter, we are going to look at the composite types in Go: arrays, slices, maps, and structs. We are also going to take another look at strings and runes and how they interact with character encodings.

Chapter 3. Errors

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Error handling is one of the biggest challenges for developers moving to Go from other languages. For those used to exceptions, Go's approach feels anachronistic. But there are solid software engineering principles underlying Go's approach. In this chapter, you'll learn how to work with errors in Go. We'll also take a look at `panic` and `recover`, Go's system for handling errors that should stop execution.

How to Handle Errors: The Basics

As was covered briefly in Chapter 5, Go handles errors by returning a value of type `error` as the last return value for a function. This is entirely by convention, but it is such a strong convention that it should never be breached. When a function executes as expected, `nil` is returned for the error parameter. If something goes wrong, an error value is returned instead. The calling function then checks the error return value by comparing it to

`nil`, handling the error, or returning an error of its own. A simple function with error handling looks like this:

```
func calcRemainderAndMod(numerator, denominator int) (int, int, error) {
    if denominator == 0 {
        return 0, 0, errors.New("denominator is 0")
    }
    return numerator / denominator, numerator % denominator, nil
}
```

A new error is created from a string by calling the `New` function in the `errors` package. Error messages should not be capitalized nor should they end with punctuation or a newline. In most cases, you should set the other return values to their zero values when a non-nil error is returned. You'll see an exception to this rule when we look at sentinel errors.

Unlike languages with exceptions, Go doesn't have special constructs to detect if an error was returned. Whenever a function returns, use an `if` statement to check the error variable to see if it is non-nil:

```
func main() {
    numerator := 20
    denominator := 3
    remainder, mod, err := calcRemainderAndMod(numerator, denominator)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Println(remainder, mod)
}
```

`error` is a built-in interface that defines a single method:

```
type error interface {
    Error() string
}
```

Anything that implements this interface is considered an error. The reason why we return `nil` from a function to indicate that no error occurred is that `nil` is the zero value for any interface type.

There are two very good reasons why Go uses a returned error instead of thrown exceptions. First, exceptions add at least one new code path through the code. These paths are sometimes unclear, especially in languages whose functions don't include a declaration that an exception is possible. This produces code that crashes in surprising ways when exceptions aren't properly handled, or, even worse, code that doesn't crash but whose data is not properly initialized, modified, or stored.

The second reason is more subtle, but demonstrates how Go's features work together. The Go compiler requires that all variables must be read. Making errors returned values forces developers to either check and handle error conditions or make it explicit that they are ignoring errors by using an underscore (_) for the returned error value.

> **NOTE**
>
> As noted in Chapter 5, while you cannot ignore *some* values returned from a function, you can ignore *all* of the return values from a function. If you ignore all the return values, you would be able to ignore the error, too. In most cases, it is very bad form to ignore the values returned from a function. Please avoid this, except for cases like `fmt.Println`.

Exception handling may produce shorter code, but having fewer lines doesn't necessarily make code easier to understand or maintain. As you've seen, idiomatic Go favors clear code, even if it takes more lines.

Another thing to note is how code flows in Go. The error handling is indented inside an `if` statement. The business logic is not. This gives a quick visual clue to which code is along the "golden path" and which code is the exceptional condition.

Use Strings for Simple Errors

Go's standard library provides two ways to create an error from a string. The first is the `errors.New` function. It takes in a `string` and returns an

`error`. This string is returned when you call the `Error` method on the returned error instance. If you pass an error to `fmt.Println`, it calls the `Error` method automatically:

```
func doubleEven(i int) (int, error) {
    if i % 2 != 0 {
        return 0, errors.New("only even numbers are processed")
    }
    return i * 2, nil
}
```

The second way is to use the `fmt.Errorf` function. This function allows you to use all of the formatting verbs for `fmt.Printf` to create an error. Like `errors.New`, this string is returned when you call the `Error` method on the returned error instance:

```
func doubleEven(i int) (int, error) {
    if i % 2 != 0 {
        return 0, fmt.Errorf("%d isn't an even number", i)
    }
    return i * 2, nil
}
```

Sentinel Errors

Some errors are meant to signal that processing cannot continue due to a problem with the current state. In his blog post "Don't just check errors, handle them gracefully", Dave Cheney, a developer who has been active in the Go community for many years, coined the term *sentinel errors* to describe these errors:

> *The name descends from the practice in computer programming of using a specific value to signify that no further processing is possible. So to [sic] with Go, we use specific values to signify an error.*
>
> —Dave Cheney

Sentinel errors are one of the few variables that are declared at the package level. By convention, their names start with `Err` (with the notable exception

of `io.EOF`). They should be treated as read-only; there's no way for the Go compiler to enforce this, but it is a programming error to change their value.

Sentinel errors are usually used to indicate that you cannot start or continue processing. For example, the standard library includes a package for processing ZIP files, `archive/zip`. This package defines several sentinel errors, including `ErrFormat`, which is returned when data that doesn't represent a ZIP file is passed in. Try out this code on The Go Playground:

```
func main() {
    data := []byte("This is not a zip file")
    notAZipFile := bytes.NewReader(data)
    _, err := zip.NewReader(notAZipFile, int64(len(data)))
    if err == zip.ErrFormat {
        fmt.Println("Told you so")
    }
}
```

Another example of a sentinel error in the standard library is `rsa.ErrMessageTooLong` in the `crypto/rsa` package. It indicates that a message cannot be encrypted because it is too long for the provided public key. The sentinel error `context.Canceled`., is covered in Chapter 14.

Be sure you need a sentinel error before you define one. Once you define one, it is part of your public API and you have committed to it being available in all future backward-compatible releases. It's far better to reuse one of the existing ones in the standard library or to define an error type that includes information about the condition that caused the error to be returned (you'll see how to do that in the next section). But if you have an error condition that indicates a specific state has been reached in your application where no further processing is possible and no additional information needs to be used to explain the error state, a sentinel error is the correct choice.

How do you test for a sentinel error? As you can see in the preceding code sample, use `==` to test if the error was returned when calling a function whose documentation explicitly says it returns a sentinel error. In an upcoming section, we will discuss how to check for sentinel errors in other situations.

USING CONSTANTS FOR SENTINEL ERRORS

In Constant errors, Dave Cheney proposed that constants would make useful sentinel errors. You'd have a type like this in a package (we'll talk about creating packages in Chapter 4):

```
package consterr

type Sentinel string

func(s Sentinel) Error() string {
    return string(s)
}
```

and then use it like this:

```
package mypkg

const (
    ErrFoo = consterr.Sentinel("foo error")
    ErrBar = consterr.Sentinel("bar error")
)
```

This looks like a function call, but it's actually casting a string literal to a type that implements the `error` interface. It would be impossible to change the values of `ErrFoo` and `ErrBar`. At first glance, this looks like a good solution.

However, this practice isn't considered idiomatic. If you used the same type to create constant errors across packages, two errors would be equal if their error strings are equal. They'd also be equal to a string literal with the same value. Meanwhile, an error created with `errors.New` is only equal to itself or to variables explicitly assigned its value. You almost certainly do not want to make errors in different packages equal to each other; otherwise, why declare two different errors? (You could avoid this by creating a nonpublic error type in every package, but that's a lot of boilerplate.)

> The sentinel error pattern is another example of the Go design philosophy. Sentinel errors should be rare, so they can be handled by convention instead of language rules. Yes, they are public package-level variables. This makes them mutable, but it's highly unlikely someone would accidentally reassign a public variable in a package. In short, it's a corner case that is handled by other features and patterns. The Go philosophy is that it's better to keep the language simple and trust the developers and tooling than it is to add features.

So far, all the errors that we've seen are strings. But Go errors can contain more information. Let's see how.

Errors Are Values

Since `error` is an interface, you can define your own errors that include additional information for logging or error handling. For example, you might want to include a status code as part of the error to indicate the kind of error that should be reported back to the user. This lets you avoid string comparisons (whose text might change) to determine error causes. Let's see how this works. First, define your own enumeration to represent the status codes:

```
type Status int

const (
    InvalidLogin Status = iota + 1
    NotFound
)
```

Next, define a `StatusErr` to hold this value:

```
type StatusErr struct {
    Status    Status
    Message string
}

func (se StatusErr) Error() string {
```

```
    return se.Message
}
```

Now we can use `StatusErr` to provide more details about what went wrong:

```
func LoginAndGetData(uid, pwd, file string) ([]byte, error) {
    err := login(uid, pwd)
    if err != nil {
        return nil, StatusErr{
            Status:    InvalidLogin,
            Message: fmt.Sprintf("invalid credentials for user %s", uid),
        }
    }
    data, err := getData(file)
    if err != nil {
        return nil, StatusErr{
            Status:    NotFound,
            Message: fmt.Sprintf("file %s not found", file),
        }
    }
    return data, nil
}
```

Even when you define your own custom error types, always use `error` as the return type for the error result. This allows you to return different types of errors from your function and allows callers of your function to choose not to depend on the specific error type.

If you are using your own error type, be sure you don't return an uninitialized instance. Let's see what happens if you do. Try out the following code on The Go Playground:

```
func GenerateError(flag bool) error {
    var genErr StatusErr
    if flag {
        genErr = StatusErr{
            Status: NotFound,
        }
    }
    return genErr
}
```

```
func main() {
    err := GenerateError(true)
    fmt.Println(err != nil)
    err = GenerateError(false)
    fmt.Println(err != nil)
}
```

Running this program produces the following output:

```
true
true
```

This isn't a pointer type versus value type issue; if we declared `genErr` to be of type `*StatusErr`, we'd see the same output. The reason why `err` is non-nil is that `error` is an interface. As we discussed in Chapter 7, for an interface to be considered `nil`, both the underlying type and the underlying value must be `nil`. Whether or not `genErr` is a pointer, the underlying type part of the interface is not `nil`.

There are two ways to fix this. The most common approach is to explicitly return `nil` for the error value when a function completes successfully:

```
func GenerateError(flag bool) error {
    if flag {
        return StatusErr{
            Status: NotFound,
        }
    }
    return nil
}
```

This has the advantage of not requiring you to read through code to make sure that the error variable on the `return` statement is correctly defined.

Another approach is to make sure that any local variable that holds an `error` is of type error:

```
func GenerateError(flag bool) error {
    var genErr error
    if flag {
        genErr = StatusErr{
```

```
                Status: NotFound,
            }
        }
        return genErr
    }
```

> **WARNING**
>
> When using custom errors, never define a variable to be of the type of your custom error. Either explicitly return `nil` when no error occurs or define the variable to be of type `error`.

As was covered in Chapter 7, don't use a type assertion or a type switch to access the fields and methods of a custom error. Instead, use `errors.As`, which is discussed in "Is and As".

Wrapping Errors

When an error is passed back through your code, you often want to add information to it. This can be the name of the function that received the error or the operation it was trying to perform. When you preserve an error while adding information, it is called *wrapping* the error. When you have a series of wrapped errors, it is called an *error tree*.

There's a function in the Go standard library that wraps errors, and we've already seen it. The `fmt.Errorf` function has a special verb, `%w`. Use this to create an error whose formatted string includes the formatted string of another error and which contains the original error as well. The convention is to write `: %w` at the end of the error format string and make the error to be wrapped the last parameter passed to `fmt.Errorf`.

The standard library also provides a function for unwrapping errors, the `Unwrap` function in the `errors` package. You pass it an error and it returns the wrapped error, if there is one. If there isn't, it returns `nil`. Here's a quick program that demonstrates wrapping with `fmt.Errorf` and unwrapping with `errors.Unwrap`. You can run it on The Go Playground:

```
func fileChecker(name string) error {
    f, err := os.Open(name)
    if err != nil {
        return fmt.Errorf("in fileChecker: %w", err)
    }
    f.Close()
    return nil
}

func main() {
    err := fileChecker("not_here.txt")
    if err != nil {
        fmt.Println(err)
        if wrappedErr := errors.Unwrap(err); wrappedErr != nil {
            fmt.Println(wrappedErr)
        }
    }
}
```

When you run this program, you see the following output:

```
in fileChecker: open not_here.txt: no such file or directory
open not_here.txt: no such file or directory
```

> **NOTE**
>
> You don't usually call `errors.Unwrap` directly. Instead, you use `errors.Is` and `errors.As` to find a specific wrapped error. We'll talk about these two functions in the next section.

If you want to wrap an error with your custom error type, your error type needs to implement the method `Unwrap`. This method takes in no parameters and returns an `error`. Here's an update to the error that we defined earlier to demonstrate how this works:

```
type StatusErr struct {
    Status Status
    Message string
    Err error
}
```

```
func (se StatusErr) Error() string {
    return se.Message
}

func (se StatusError) Unwrap() error {
    return se.Err
}
```

Now we can use `StatusErr` to wrap underlying errors:

```
func LoginAndGetData(uid, pwd, file string) ([]byte, error) {
    err := login(uid,pwd)
    if err != nil {
        return nil, StatusErr {
            Status: InvalidLogin,
            Message: fmt.Sprintf("invalid credentials for user %s",uid),
            Err: err,
        }
    }
    data, err := getData(file)
    if err != nil {
        return nil, StatusErr {
            Status: NotFound,
            Message: fmt.Sprintf("file %s not found",file),
            Err: err,
        }
    }
    return data, nil
}
```

Not all errors need to be wrapped. A library can return an error that means processing cannot continue, but the error message contains implementation details that aren't needed in other parts of your program. In this situation it is perfectly acceptable to create a brand-new error and return that instead. Understand the situation and determine what needs to be returned.

TIP

If you want to create a new error that contains the message from another error, but don't want to wrap it, use `fmt.Errorf` to create an error, but use the `%v` verb instead of `%w`:

```
err := internalFunction()
if err != nil {
    return fmt.Errorf("internal failure: %v", err)
}
```

Wrapping Multiple Errors

Sometimes a function generates multiple errors that should be returned. For example, if you wrote a function to validate the fields in a struct, it would be better to return an error for each invalid field. Since the standard function signature returns `error` and not `[]error`, you need to merge multiple errors into a single error. That's what the `errors.Join` function is for.

```
type Person struct {
    FirstName string
    LastName string
    Age int
}

func ValidatePerson(p Person) error {
    var errors []error
    if len(p.FirstName) == 0 {
        errors = append(errors, errors.New("field FirstName cannot be empty"))
    }
    if len(p.LastName) == 0 {
        errors = append(errors, errors.New("field LastName cannot be empty"))
    }
    if p.Age < 0 {
        errors = append(errors, errors.New("field Age cannot be negative"))
    }
    if len(errors) > 0 {
        return errors.Join(errors...)
    }
```

```
    return nil
}
```

Another way to merge multiple errors is to pass multiple `%w` verbs to `fmt.Errorf`:

```
err1 := errors.New("first error")
err2 := errors.New("second error")
err3 := errors.New("third error")
err := fmt.Errorf("first: %w, second: %w, third: %w", err1, err2, err3)
```

You can implement your own `error` type that supports multiple wrapped errors. To do so, implement the `Unwrap` method, but have it return `[]error` instead of `error`.

```
type MyError struct {
    Code int
    Errors []error
}

func (m MyError) Unwrap() []error {
    return m.Errors
}
```

Go doesn't support method overloading, so you can't create a single type that provides both implementations of `Unwrap`. Also note that the `errors.Unwrap` function will return `nil` if you pass it an error that implements the `[]error` variant of `Unwrap`. This is another reason why `errors.Unwrap` isn't terribly useful.

If you need to handle errors that may wrap zero, one, or multiple errors, use this code as a basis:

```
var err error
err = funcThatReturnsAnError()
switch err := err.(type) {
case interface {Unwrap() error}:
    // handle single error
    innerErr := err.Unwrap()
    // process innerErr
case interface {Unwrap() []error}:
```

```
        //handle multiple wrapped errors
        innerErrs := err.Unwrap()
        for _, innerErr := range innerErrs {
            // process each innerErr
        }
    default:
        // handle no wrapped error
    }
```

Since the standard library doesn't define interfaces to represent errors with either `Unwrap` variant, this code uses anonymous interfaces in a type switch to match the methods and access the wrapped errors. Before writing your own code, see if you can use `errors.Is` and `errors.As` to examine your error trees. Let's see how they work.

Is and As

Wrapping errors is a useful way to get additional information about an error, but it introduces problems. If a sentinel error is wrapped, you cannot use `==` to check for it, nor can you use a type assertion or type switch to match a wrapped custom error. Go solves this problem with two functions in the `errors` package, `Is` and `As`.

To check if the returned error or any errors that it wraps match a specific sentinel error instance, use `errors.Is`. It takes in two parameters, the error being checked and the instance you are comparing it against. The `errors.Is` function returns `true` if there is an error in the error tree that matches the provided sentinel error. Let's write a short program to see `errors.Is` in action. You can run it yourself on The Go Playground:

```
func fileChecker(name string) error {
    f, err := os.Open(name)
    if err != nil {
        return fmt.Errorf("in fileChecker: %w", err)
    }
    f.Close()
    return nil
}
```

```
func main() {
    err := fileChecker("not_here.txt")
    if err != nil {
        if errors.Is(err, os.ErrNotExist) {
            fmt.Println("That file doesn't exist")
        }
    }
}
```

Running this program produces the output:

```
That file doesn't exist
```

By default, `errors.Is` uses `==` to compare each wrapped error with the specified error. If this does not work for an error type that you define (for example, if your error is a noncomparable type), implement the `Is` method on your error:

```
type MyErr struct {
    Codes []int
}

func (me MyErr) Error() string {
    return fmt.Sprintf("codes: %v", me.Codes)
}

func (me MyErr) Is(target error) bool {
    if me2, ok := target.(MyErr); ok {
        return reflect.DeepEqual(me, me2)
    }
    return false
}
```

(The `reflect.DeepEqual` function was mentioned back in Chapter 3. It can compare anything, including slices.)

Another use for defining your own `Is` method is to allow comparisons against errors that aren't identical instances. You might want to pattern match your errors, specifying a filter instance that matches errors that have some of the same fields. Let's define a new error type, `ResourceErr`:

```
type ResourceErr struct {
    Resource     string
    Code         int
}

func (re ResourceErr) Error() string {
    return fmt.Sprintf("%s: %d", re.Resource, re.Code)
}
```

If you want two `ResourceErr` instances to match when either field is set, you can do so by writing a custom `Is` method:

```
func (re ResourceErr) Is(target error) bool {
    if other, ok := target.(ResourceErr); ok {
        ignoreResource := other.Resource == ""
        ignoreCode := other.Code == 0
        matchResource := other.Resource == re.Resource
        matchCode := other.Code == re.Code
        return matchResource && matchCode ||
            matchResource && ignoreCode ||
            ignoreResource && matchCode
    }
    return false
}
```

Now we can find, for example, all errors that refer to the database, no matter the code:

```
if errors.Is(err, ResourceErr{Resource: "Database"}) {
    fmt.Println("The database is broken:", err)
    // process the codes
}
```

You can see this code on The Go Playground.

The `errors.As` function allows you to check if a returned error (or any error it wraps) matches a specific type. It takes in two parameters. The first is the error being examined and the second is a pointer to a variable of the type that you are looking for. If the function returns `true`, an error in the error tree was found that matched, and that matching error is assigned to the

second parameter. If the function returns `false`, no match was found in the error tree. Let's try it out with `MyErr`:

```
err := AFunctionThatReturnsAnError()
var myErr MyErr
if errors.As(err, &myErr) {
    fmt.Println(myErr.Code)
}
```

Note that you use `var` to declare a variable of a specific type set to the zero value. You then pass a pointer to this variable into `errors.As`.

You don't have to pass a pointer to a variable of an error type as the second parameter to `errors.As`. You can pass a pointer to an interface to find an error that meets the interface:

```
err := AFunctionThatReturnsAnError()
var coder interface {
    Code() int
}
if errors.As(err, &coder) {
    fmt.Println(coder.Code())
}
```

The example uses an anonymous interface, but any interface type is acceptable.

> **WARNING**
>
> If the second parameter to `errors.As` is anything other than a pointer to an error or a pointer to an interface, the method panics.

Just like you can override the default `errors.Is` comparison with an `Is` method, you can override the default `errors.As` comparison with an `As` method on your error. Implementing an `As` method is nontrivial and requires reflection (we will talk about reflection in Go in Chapter 16). You should only do it in unusual circumstances, such as when you want to match an error of one type and return another.

> **TIP**
>
> Use `errors.Is` when you are looking for a specific *instance* or specific *values*. Use `errors.As` when you are looking for a specific *type*.

Wrapping Errors with defer

Sometimes you find yourself wrapping multiple errors with the same message:

```
func DoSomeThings(val1 int, val2 string) (string, error) {
    val3, err := doThing1(val1)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    val4, err := doThing2(val2)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    result, err := doThing3(val3, val4)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    return result, nil
}
```

You can simplify this code by using `defer`:

```
func DoSomeThings(val1 int, val2 string) (_ string, err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("in DoSomeThings: %w", err)
        }
    }()
    val3, err := doThing1(val1)
    if err != nil {
        return "", err
    }
    val4, err := doThing2(val2)
    if err != nil {
        return "", err
    }
```

```
    return doThing3(val3, val4)
}
```

You have to name the return values so that you can refer to `err` in the deferred function. If you name a single return value, you must name all of them, so you use an underscore here for the string return value that isn't explicitly assigned.

In the `defer` closure, the code checks if an error was returned. If so, it reassigns the error to a new error that wraps the original error with a message that indicates which function detected the error.

This pattern works well when you are wrapping every error with the same message. If you want to customize the wrapping error with more detail, then put both the specific and the general message in every `fmt.Errorf`.

panic and recover

Previous chapters have mentioned panics in passing without going into any details on what they are. A panic is similar to an `Error` in Java or Python. It is a state generated by the Go runtime whenever it is unable to figure out what should happen next. This could be due to a programming error (like an attempt to read past the end of a slice) or environmental problem (like running out of memory). As soon as a panic happens, the current function exits immediately and any defers attached to the current function start running. When those defers complete, the defers attached to the calling function run, and so on, until `main` is reached. The program then exits with a message and a stack trace.

If there are situations in your programs that are unrecoverable, you can create your own panics. The built-in function `panic` takes one parameter, which can be of any type. Usually, it is a string. Let's make a trivial program that panics and run it on The Go Playground:

```
func doPanic(msg string) {
    panic(msg)
}
```

```
func main() {
    doPanic(os.Args[0])
}
```

Running this code produces the following output:

```
panic: /tmpfs/play

goroutine 1 [running]:
main.doPanic(...)
    /tmp/sandbox567884271/prog.go:6
main.main()
    /tmp/sandbox567884271/prog.go:10 +0x5f
```

As you can see, a `panic` prints out its message followed by a stack trace.

Go provides a way to capture a panic to provide a more graceful shutdown or to prevent shutdown at all. The built-in `recover` function is called from within a `defer` to check if a panic happened. If there was a panic, the value assigned to the panic is returned. Once a `recover` happens, execution continues normally. Let's take a look with another sample program. Run it on The Go Playground:

```
func div60(i int) {
    defer func() {
        if v := recover(); v != nil {
            fmt.Println(v)
        }
    }()
    fmt.Println(60 / i)
}

func main() {
    for _, val := range []int{1, 2, 0, 6} {
        div60(val)
    }
}
```

There's a specific pattern for using `recover`. We register a function with `defer` to handle a potential `panic`. We call `recover` within an `if` statement and check to see if a non-nil value was found. You must call `recover` from

within a `defer` because once a panic happens, only deferred functions are run.

Running this code produces the following output:

```
60
30
runtime error: integer divide by zero
10
```

While `panic` and `recover` look a lot like exception handling in other languages, they are not intended to be used that way. Reserve panics for fatal situations and use `recover` as a way to gracefully handle these situations. If your program panics, be very careful about trying to continue executing after the panic. It's very rare that you want to keep your program running after a panic occurs. If the panic was triggered because the computer is out of a resource like memory or disk space, the safest thing to do is use `recover` to log the situation to monitoring software and shut down with `os.Exit(1)`. If there's a programming error that caused the panic, you can try to continue, but you'll likely hit the same problem again. In the preceding sample program, it would be idiomatic to check for division by zero and return an error if one was passed in.

The reason we don't rely on `panic` and `recover` is that `recover` doesn't make clear *what* could fail. It just ensures that *if* something fails, we can print out a message and continue. Idiomatic Go favors code that explicitly outlines the possible failure conditions over shorter code that handles anything while saying nothing.

There is one situation where `recover` is recommended. If you are creating a library for third parties, do not let panics escape the boundaries of your public API. If a panic is possible, a public function should use a `recover` to convert the `panic` into an error, return it, and let the calling code decide what to do with them.

> **NOTE**
>
> While the HTTP server built into Go recovers from panics in handlers, David Symonds said in a GitHub comment that as of 2015, this is now considered a mistake by the Go team

Getting a Stack Trace from an Error

One of the reasons why new Go developers are tempted to use `panic` and `recover` is they want to get a stack trace when something goes wrong. By default, Go doesn't provide that. As we've shown, you can use error wrapping to build a call stack by hand, but there are third-party libraries with error types that generate those stacks automatically (see Chapter 4 to learn how to incorporate third-party code in your program). The best known third-party library provides functions for wrapping errors with stack traces.

By default, the stack trace is not printed out. If you want to see the stack trace, use `fmt.Printf` and the verbose output verb (`%+v`). Check the documentation to learn more.

Despite the popularity of `pkg/errors`, it is no longer being maintained and has been archived. If you are uncomfortable using an unsupported library, I have a similar one.

> **NOTE**
>
> When you have a stack trace in your error, the output includes the full path to the file on the computer where the program was compiled. If you don't want to expose the path, use the `-trimpath` flag when building your code. This replaces the full path with the package.

Exercises

Look at the code in github.com/learning-go-book-2e/sample_code/exercise. You are going to modify this code in each of these exercises. It works

correctly, but improvements should be made to its error handling.

1. Create a sentinel error to represent an invalid ID. In `main()`, use `errors.Is` to check for the sentinel error, and print a message when it is found.
2. Define a custom error type to represent an empty field error. This error should include the name of the empty `Employee` field. In `main()`, Use `errors.As` to check for this error. Print out a message that includes the field name.
3. Rather than returning the first error found, return back a single error that contains all errors discovered during validation. Update the code in `main()` to properly report multiple errors.

Wrapping Up

This chapter covered errors in Go, what they are, how to define your own, and how to examine them. We also took a look at `panic` and `recover`. The next chapter discusses packages and modules, how to use third-party code in your programs, and how to publish your own code for others to use.

Chapter 4. Modules, Packages, and Imports

Most modern programming languages have a system for organizing code into namespaces and libraries, and Go is no exception. As you've seen while exploring other features, Go introduces some new approaches to this old idea. In this chapter, you'll learn how to organize code with packages and modules, how to import them, how to work with third-party libraries, and how to create libraries of your own.

Repositories, Modules, and Packages

Library management in Go is based around three concepts: *repositories*, *modules*, and *packages*. A repository is familiar to all developers. It is a place in a version control system where the source code for a project is stored. A module is a bundle of Go source code that's distributed and versioned as a single unit. Modules are stored in a repository. Modules

consist of one or more packages, which are directories of source code. Packages give a module organization and structure.

> **NOTE**
>
> While you can store more than one module in a repository, it is discouraged. Everything within a module is versioned together. Maintaining two modules in one repository requires you to track separate versions for two different modules in a single repository.

Before using code from packages outside of the standard library, you need to make sure that you have a properly created module. Every module has a globally unique identifier. This is not unique to Go. Java uses globally unique package declarations like `com.companyname.projectname.library`.

In Go, this name is called a *module path*. It is usually based on the repository where the module is stored. For example, Proteus, a module I wrote to simplify relational database access in Go, can be found at *https://github.com/jonbodner/proteus*. It has a module path of *github.com/jonbodner/proteus*.

> **NOTE**
>
> Back in "Your First Go Program", we created a module whose name was `hello_world`, which is obviously not globally unique. This is fine if you are creating a module for local use only. If you put a module with a non-unique name into a source code repository, it cannot be imported by another module.

go.mod

A directory of Go source code becomes a module when there's a valid *go.mod* file in it. Rather than create this file manually, we use the subcommands of the `go mod` command to manage modules. The command `go mod init` *`MODULE_PATH`* creates the *go.mod* file that makes the

current directory the root of a module. The *MODULE_PATH* is the globally unique name that identifies your module. The module path is case-sensitive. To reduce confusion, do not use uppercase letters within it.

Let's take a look at the contents of a *go.mod* file:

```
module github.com/learning-go-book-2e/money

go 1.21

require (
    github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-18...
    github.com/shopspring/decimal v1.3.1
)

require (
    github.com/fatih/color v1.13.0 // indirect
    github.com/mattn/go-colorable v0.1.9 // indirect
    github.com/mattn/go-isatty v0.0.14 // indirect
    golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c // indirect
)
```

Every *go.mod* file starts with a `module` directive that consists of the word `module` and the module's unique path. Next, the *go.mod* file specifies the minimum compatible version of Go with the `go` directive. All source code within the module must be compatible with the specified version. For example, if you specify the (rather old) version `1.12`, the compiler will not let you use underscores within numeric literals, because that feature was added in Go 1.13. The Go version also controls features of the Go build tools. The tool behavior described in this book matches Go 1.21, the latest version at the time the book was written. If you need to work with a module that specifies an older version, you might see minor differences.

The next sections in a `go.mod` file are the `require` directives. The `require` directives are only present if your module has dependencies. They list the modules that your module depends on and the minimum version required for each one. The first `require` section lists the direct dependencies of your module. The second one lists the dependencies of the dependencies of your module. Each line in this section ends with an `// indirect` comment.

There's no functional difference between modules marked as indirect and those that aren't; it's just documentation for people when they look at the `go.mod` file. There is one situation where direct dependencies are marked as indirect, and we will discuss it when we talk about the different ways to use `go get`. In "Importing Third-Party Code", you'll learn more about adding and managing the dependencies of your module.

While the `module`, `go`, and `require` directives are the ones most commonly used in a `go.mod` file, there are three others directives as well. We will cover the `replace` and `exclude` directives in "Overriding dependencies", and we'll cover the `retract` directive in "Retracting a version of your module".

Building Packages

Now that you've learned how to make our directory of code into a module, it's time to start using packages to organize our code. We'll start by looking at how `import` works, move on to creating and organizing packages, and then look at some of the features of Go's packages, both good and bad.

Imports and Exports

The example programs have been using the `import` statement in Go without discussing what it does and how importing in Go differs from other languages. Go's `import` statement allows you to access exported constants, variables, functions, and types in another package. A package's exported identifiers (an *identifier* is the name of a variable, constant, type, function, method, or a field in a struct) cannot be accessed from another current package without an `import` statement.

This leads to the question: how do you export an identifier in Go? Rather than use a special keyword, Go uses *capitalization* to determine if a package-level identifier is visible outside of the package where it is declared. An identifier whose name starts with an uppercase letter is *exported*. Conversely, an identifier whose name starts with a lowercase

letter or underscore can only be accessed from within the package where it is declared.

Anything you export is part of your package's API. Before you export an identifier, be sure that you intend to expose it to clients. Document all exported identifiers and keep them backward-compatible unless you are intentionally making a major version change (see "Versioning Your Module" for more information).

Creating and Accessing a Package

Making packages in Go is easy. Let's look at a small program to demonstrate this. You can find it on GitHub. Inside `package_example`, you'll see two additional directories, *math* and *formatter*. In *math*, there's a file called *math.go* with the following contents:

```
package math

func Double(a int) int {
    return a * 2
}
```

The first line of the file is called the *package clause*. It consists of the keyword `package` and the name for the package. The package clause is always the first nonblank, noncomment line in a Go source file.

In the *do-format* directory, there's a file called *formatter.go* with the following contents:

```
package format

import "fmt"

func Number(num int) string {
    return fmt.Sprintf("The number is %d", num)
}
```

Note that the package name is `format` in the package clause, but it's in the do-format directory. How to interact with this package will be covered very

shortly.

Finally, the following contents are in the file *main.go* in the root directory:

```
package main

import (
    "fmt"

    "github.com/learning-go-book-2e/package_example/do-format"
    "github.com/learning-go-book-2e/package_example/math"
)

func main() {
    num := math.Double(2)
    output := format.Number(num)
    fmt.Println(output)
}
```

The first line of this file is familiar. All of our programs before this chapter have put `package main` as the first line in our code. We'll talk more about what this means in just a bit.

Next we have our import section. It imports three packages. The first is `fmt`, which is in the standard library. We've done this in previous chapters. The next two imports refer to the packages within our program. You must specify an *import path* when importing from anywhere besides the standard library. The import path is built by appending the path to the package within the module to the module path.

It is a compile-time error to import a package but not use any of the identifiers exported by the package. This ensures that the binary produced by the Go compiler only includes code that's actually used in the program.

> **WARNING**
>
> While you can use a relative path to import a dependent package within the same module, don't do this. Absolute import paths clarify what you are importing and make it easier to refactor your code. You must fix the imports when a file with a relative path in its imports is moved to another package, and if you move that file to another module entirely, you must make the import reference absolute.

When you run this program, you'll see the following output:

```
$ go build
$ ./package_example
The number is 4
```

The `main` function called the `Double` function in the `math` package by prefixing the function name with the package name. We've seen this in previous chapters when calling functions in the standard library. We also called the `Number` function in the `format` package. You might wonder where this `format` package came from, since the import says `github.com/learning-go-book-2e/package_example/do-format`.

Every Go file in a directory must have an identical package clause. (There is one tiny exception to this rule that you'll see in Chapter 15.) We imported the `format` package with the import path *github.com/learning-go-book-2e/package_example/do-format*. That's because *the name of a package is determined by its package clause, not its import path*.

As a general rule, you should make the name of the package match the name of the directory that contains the package. It is hard to discover a package's name if it does not match the containing directory. However, there are a few situations where you use a different name for the package than for the directory.

The first is something we have been doing all along without realizing it. We declare a package to be a starting point for a Go application by using the special package name `main`. Since you cannot import the `main` package, this doesn't produce confusing import statements.

The other reasons for having a package name not match your directory name are less common. If your directory name contains a character that's not valid in a Go identifier, then you must choose a package name that's different from your directory name. In our case, `do-format` is not a valid identifier name, so it's replaced with `format`. It's better to avoid this by never creating a directory with a name that's not a valid identifier.

The final reason for creating a directory whose name doesn't match the package name is to support versioning using directories. We'll talk about this more in "Versioning Your Module".

Package names are in the file block. If you use the same package in two different files in the same package, you must import the package in both files.

Naming Packages

Package names should be descriptive. Rather than have a package called `util`, create a package name that describes the functionality provided by the package. For example, say you have two helper functions: one to extract all names from a string and another to format names properly. Don't create two functions in a `util` package called `ExtractNames` and `FormatNames`. If you do, every time you use these functions, they will be referred to as `util.ExtractNames` and `util.FormatNames`, and that `util` package tells you nothing about what the functions do.

One option is to create one function called `Names` in a package called `extract` and a second function called `Names` in a package called `format`. It's OK for these two functions to have the same name, because they will always be disambiguated by their package names. The first will be referred to as `extract.Names` when imported, and the second will be referred to as `format.Names`.

An even better option is to think about parts of speech. A function or method does something, so it should be a verb or action word. A package would be a noun, a name for the kind of item that is created or modified by the functions in the package. By following these rules, you would create a

package called `names` with two functions, `Extract` and `Format`. The first would then be referred to as `names.Extract` and the second would be `names.Format`.

You should also avoid repeating the name of the package in the names of functions and types within the package. Don't name your function `ExtractNames` when it is in the `names` package. The exception to this rule is when the name of the identifier is the same as the name of the package. For example, the package `sort` in the standard library has a function called `Sort`, and the `context` package defines the `Context` interface.

Overriding a Package's Name

Sometimes you might find yourself importing two packages whose names collide. For example, the standard library includes two packages for generating random numbers; one is cryptographically secure (`crypto/rand`) and the other is not (`math/rand`). The regular generator is fine when you aren't generating random numbers for encryption, but you need to seed it with an unpredictable value. A common pattern is to seed a regular random number generator with a value from a cryptographic generator. In Go, both packages have the same name (`rand`). When that happens, you provide an alternate name for one package within the current file. You can try out this code on The Go Playground. First, look at the import section:

```
import (
    crand "crypto/rand"
    "encoding/binary"
    "fmt"
    "math/rand"
)
```

We import `crypto/rand` with the name `crand`. This overrides the name `rand` that's declared within the package. We then import `math/rand` normally. When you look at the `seedRand` function, you see that we access identifiers in `math/rand` with the `rand` prefix, and use the `crand` prefix with the `crypto/rand` package:

```
func seedRand() *rand.Rand {
    var b [8]byte
    _, err := crand.Read(b[:])
    if err != nil {
        panic("cannot seed with cryptographic random number generator")
    }
    r := rand.New(rand.NewSource(int64(binary.LittleEndian.Uint64(b[:]))))
    return r
}
```

> **NOTE**
>
> There are two other symbols you can use as a package name. The package name `.` places all the exported identifiers in the imported package into the current package's namespace; you don't need a prefix to refer to them. This is discouraged because it makes your source code less clear. You can no longer tell if something is defined in the current package or if it was imported by simply looking at its name.
>
> You can also use _ as the package name. We'll explore what this does when we talk about `init` in "The init Function: Avoid if Possible".

As we discussed in Chapter 4, package names can be shadowed. Declaring variables, types, or functions with the same name as a package makes the package inaccessible within the block with that declaration. If this is unavoidable (for example, a newly imported package has a name that conflicts with an existing identifier), override the package's name to resolve the conflict.

Create Module Documentation with godoc

An important part of creating a module for others to use is documenting it properly. Go has its own format for writing comments that are automatically converted into documentation. It's called *godoc* format, and it's very simple. Here are the rules:

- Place the comment directly before the item being documented with no blank lines between the comment and the declaration of the item.

- Start each line of the comment with double slashes (//), followed by a space. While it's legal to use /* and */ to mark your comment block, it is idiomatic to use double slashes.
- The first word in the comment for a symbol (a function, type, constant, variable, or method) should be the name of the symbol. You can also use "A" or "An" before the symbol name, if it helps make the comment text grammatically correct.
- Use a blank comment line (double slashes and a newline) to break your comment into multiple paragraphs.

As we'll talk about in "pkg.go.dev", you can view public documentation online in HTML format. If you want to make your documents look a little snazzier, there are a couple of ways to format it:

- If you want your comment to contain some preformatted content (such as a table or source code), put an additional space after the double slashes to indent the lines with the content.
- If you want a header in your comment, put a # and a space after the double slashes . Unlike Markdown, you cannot use multiple # characters to make different levels of headers.
- To make a link to another package (whether or not it is in the current module), put the package path within brackets ([and]).
- If you include a raw URL in your comment, it will be converted into a link.
- If you want to include text that links to a web page, put the text within brackets ([and]). At the end of the comment block, declare the mappings between your text and their URLs with the format // [TEXT]: URL. You'll see a sample of this in a moment.

Comments before the package declaration create package-level comments. If you have lengthy comments for the package (such as the extensive

formatting documentation in the `fmt` package), the convention is to put the comments in a file in your package called *doc.go*.

Let's go through a well-commented file, starting with the package-level comment in Example 4-1.

Example 4-1. A package-level comment

```
// money provides various utilities to make it easy to manage money.
package money
```

Next, we place a comment on an exported struct (see Example 4-2). Notice that it starts with the name of the struct.

Example 4-2. A struct comment

```
// Money represents the combination of an amount of money
// and the currency the money is in.
//
// The value is stored using a [github.com/shopspring/decimal.Decimal]
type Money struct {
    Value decimal.Decimal
    Currency string
}
```

Finally, we have a comment on a function (see Example 4-3).

Example 4-3. A well-commented function

```
// Convert converts the value of one currency to another.
//
// It has two parameters: a Money instance with the value to convert,
// and a string that represents the currency to convert to. Convert returns
// the converted currency and any errors encountered from unknown or
unconvertible
// currencies.
//
// If an error is returned, the Money instance is set to the zero value.
//
// Supported currencies are:
//        USD - US Dollar
//        CAD - Canadian Dollar
//        EUR - Euro
//        INR - Indian Rupee
//
// More information on exchange rates can be found at [Investopedia].
//
// [Investopedia]: https://www.investopedia.com/terms/e/exchangerate.asp
```

```
func Convert(from Money, to string) (Money, error) {
    // ...
}
```

Go includes a command-line tool called `go doc` that displays godocs. The command `go doc` *`PACKAGE_NAME`* displays the documentation for the specified package and a list of the identifiers in the package. Use `go doc` *`PACKAGE_NAME.IDENTIFIER_NAME`* to display the documentation for a specific identifier in the package.

Unfortunately, Go doesn't include any tools to let you preview your documentation's HTML formatting before it is published on the web. I wrote a tool that lets you take a look at what your documentation looks before it's published. It can also generate static HTML of your code's comments, and supports Markdown output as well.

You can find even more details about comments and potential pitfalls by reading through the official Go Doc Comments documentation.

> **TIP**
>
> Make sure you comment your code properly. At the very least, any exported identifier should have a comment. In "Code-quality scanners" we will look at some third-party tools that can report missing comments on exported identifiers.

The internal Package

Sometimes you want to share a function, type, or constant between packages in your module, but you don't want to make it part of your API. Go supports this via the special `internal` package name.

When you create a package called `internal`, the exported identifiers in that package and its subpackages are only accessible to the direct parent package of `internal` and the sibling packages of `internal`. Let's look at an example to see how this works. You can find the code on GitHub. The directory tree is shown in Figure 4-1.

We've declared a simple function in the *internal.go* file in the `internal` package:

```
func Doubler(a int) int {
    return a * 2
}
```

We can access this function from *foo.go* in the `foo` package and from *sibling.go* in the `sibling` package.

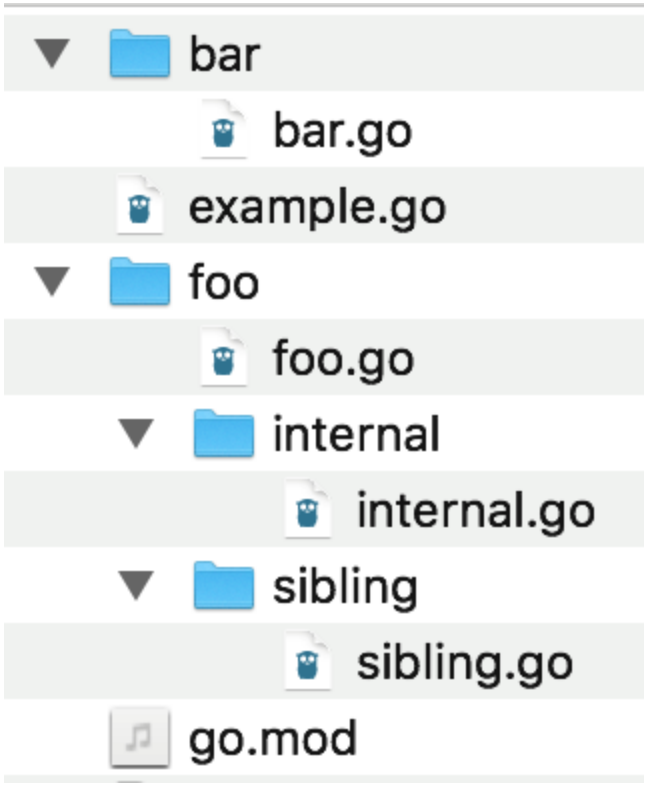

Figure 4-1. The file tree for `internal_package_example`

Be aware that attempting to use the internal function from *bar.go* in the `bar` package or from *example.go* in the root package results in a compilation error:

```
$ go build ./...
package github.com/learning-go-book-2e/internal_example
example.go:3:8: use of internal package
github.com/learning-go-book-2e/internal_example/foo/internal not allowed
```

```
package github.com/learning-go-book-2e/internal_example/bar
bar/bar.go:3:8: use of internal package
github.com/learning-go-book-2e/internal_example/foo/internal not allowed
```

Circular Dependencies

Two of the goals of Go are a fast compiler and easy to understand source code. To support this, Go does not allow you to have a *circular dependency* between packages. This means that if package A imports package B, directly or indirectly, package B cannot import package A, directly or indirectly. Let's look at a quick example to explain the concept. You can download the code from GitHub. Our module has two sub-directories, *pet* and *person*. In *pet.go* in the `pet` package, we import `github.com/learning-go-book-2e/circular_dependency_example/person`:

```
var owners = map[string]person.Person{
    "Bob":     {"Bob", 30, "Fluffy"},
    "Julia": {"Julia", 40, "Rex"},
}
```

While in *person.go* in the `person` package, we import `github.com/learning-go-book-2e/circular_dependency_example/pet`:

```
var pets = map[string]pet.Pet{
    "Fluffy": {"Fluffy", "Cat", "Bob"},
    "Rex":    {"Rex", "Dog", "Julia"},
}
```

If you try to build this module, you'll get an error:

```
$ go build
package github.com/learning-go-book-2e/circular_dependency_example
    imports github.com/learning-go-book-2e/circular_dependency_example/person
    imports github.com/learning-go-book-2e/circular_dependency_example/pet
    imports github.com/learning-go-book-2e/circular_dependency_example/person:
        import cycle not allowed
```

If you find yourself with a circular dependency, you have a few options. In some cases, this is caused by splitting packages up too finely. If two packages depend on each other, there's a good chance they should be merged into a single package. We can merge our `person` and `pet` packages into a single package and that solves our problem.

If you have a good reason to keep your packages separated, it may be possible to move just the items that cause the circular dependency to one of the two packages or to a new package.

How to Organize Your Module

There's no official way to structure the Go packages in your module, but several patterns have emerged over the years. They are guided by the principle that you should focus on making your code easy to understand and maintain.

When your module is small, keep all of your code in a single package. As long as there are no other modules that depend on your module, there is no harm in delaying organization.

As your module grows, you'll want to impose some order to make your code more readable. The first question to ask is what type of module are you creating. You can group modules into two broad categories: those that are intended as a single application and those that are primarily intended as libraries. If your are sure that your module is only intended to be used as an application, make the root of the project the `main` package. The code in the main package should be minimal; place all of your logic in an `internal` directory and the code in the `main` function will simply invoke code within `internal` This way, you can ensure that no one is going to create a module that depends on your application's implementation.

If you want your module to be used as a library, the root of your module should have a package name that matches the repository name. This makes sure that the import name matches the package name. It's not uncommon for library modules to have one or more applications included with them as utilities. In this case, create a directory called *cmd* at the root of your

module. Within *cmd*, create one directory for each binary built from your module. For example, you might have a module that contains both a web application and a command-line tool that analyzes data in the web application's database. Use `main` as the package name within each of these directories.

For more detailed information, this blog post by Eli Bendersky provides good advice on how you should structure a simple Go module.

As projects get more complicated, you'll be tempted to break up your packages. Make sure to organize your code to limit the dependencies between packages. One common pattern is to organize your code by slices of functionality. For example, if you wrote a shopping site in Go, you might place all of the code for customer management in one package and all of the code for inventory management in another. This style limits the dependencies between packages, which makes it easier to later refactor a single web application into multiple microservices. This style is in contrast to the way that many Java applications are organized, with all of the business logic in one package, all of the database logic in another package, and the data transfer objects in a third.

When developing a library, take advantage of the `internal` package. If you create multiple packages within a module and they are outside of an `internal` package, exporting a symbol so it can be used by another package in your module means it can also be used by *anyone* who imports your module. There's a principle in software engineering called Hyrum's Law: "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody." Once something is part of your API, you have a responsibly to continue supporting it until you decide to make a new version that's not backwards compatible. You will learn how to do this in "Updating to Incompatible Versions". If you have some symbols that you only want to share within your module, put them in `internal`. If you change your mind, you can always move the package out of `internal` later.

For a good overview of Go project structure advice, watch Kat Zien's talk from GopherCon 2018, How Do You Structure Your Go Apps.

> **WARNING**
>
> The "golang-standards" GitHub repository claims to be the "standard" module layout. Russ Cox, the development lead for Go, has publically stated that it is not endorsed by the Go team, and that the structure it recommends is in fact an anti-pattern. Please do not cite this repository as a way to organize your code.

Gracefully Renaming and Reorganizing Your API

After using a module for a while, you might realize that its API is not ideal. You might want to rename some of the exported identifiers or move them to another package within your module. To avoid a backward-breaking change, don't remove the original identifiers; provide an alternate name instead.

This is easy with a function or method. You declare a function or method that calls the original. For a constant, simply declare a new constant with the same type and value, but a different name.

If you want to rename or move an exported type, you use an *alias*. Quite simply, an alias is a new name for a type. We saw in Chapter 7 how to use the `type` keyword to declare a new type based on an existing one. We also use the `type` keyword to declare an alias. Let's say we have a type called `Foo`:

```
type Foo struct {
    x int
    S string
}

func (f Foo) Hello() string {
    return "hello"
}

func (f Foo) goodbye() string {
```

```
    return "goodbye"
}
```

If we want to allow users to access `Foo` by the name `Bar`, all we need to do is:

```
type Bar = Foo
```

To create an alias, we use the `type` keyword, the name of the alias, an equals sign, and the name of the original type. The alias has the same fields and methods as the original type.

The alias can even be assigned to a variable of the original type without a type conversion:

```
func MakeBar() Bar {
    bar := Bar{
        x: 20,
        S: "Hello",
    }
    var f Foo = bar
    fmt.Println(f.Hello())
    return bar
}
```

One important point to remember: an alias is just another name for a type. If you want to add new methods or change the fields in an aliased struct, you must add them to the original type.

You can alias a type that's defined in the same package as the original type or in a different package. You can even alias a type from another module. There is one drawback to an alias in another package: you cannot use an alias to refer to the unexported methods and fields of the original type. This limitation makes sense, as aliases exist to allow a gradual change to a package's API, and the API only consists of the exported parts of the package. To work around this limitation, call code in the type's original package to manipulate unexported fields and methods.

There are two kinds of exported identifiers that can't have alternate names. The first is a package-level variable. The second is a field in a struct. Once you choose a name for an exported struct field, there's no way to create an alternate name.

The init Function: Avoid if Possible

When you read Go code, it is usually clear which methods and functions are called. One of the reasons why Go doesn't have method overriding or function overloading is to make it easier to understand what code is running. However, there is a way to set up state in a package without explicitly calling anything: the `init` function. When you declare a function named `init` that takes no parameters and returns no values, it runs the first time the package is referenced by another package. Since `init` functions do not have any inputs or outputs, they can only work by side effect, interacting with package-level functions and variables.

The `init` function has another unique feature. Go allows you to declare multiple `init` functions in a single package, or even in a single file in a package. There's a documented order for running multiple `init` functions in a single package, but rather than remembering it, it's better to simply avoid them.

Some packages, like database drivers, use `init` functions to register the database driver. However, you don't use any of the identifiers in the package. As mentioned earlier, Go doesn't allow you to have unused imports. To work around this, Go allows *blank imports*, where the name assigned to an import is the underscore (_). Just as an underscore allows you to skip an unused return value from a function, a blank import triggers the `init` function in a package but doesn't give you access to any of the exported identifiers in the package:

```
import (
    "database/sql"

    _ "github.com/lib/pq"
)
```

This pattern is considered obsolete because it's unclear that a registration operation is being performed. Go's compatibility guarantee for its standard library means that we are stuck using it to register database drivers and image formats, but if you have a registry pattern in your own code, register your plug-ins explicitly.

The primary use of `init` functions today is to initialize package-level variables that can't be configured in a single assignment. It's a bad idea to have mutable state at the top level of a package, since it makes it harder to understand how data flows through your application. That means that any package-level variables configured via `init` should be *effectively immutable*. While Go doesn't provide a way to enforce that their value does not change, you should make sure that your code does not change them. If you have package-level variables that need to be modified while your program is running, see if you can refactor your code to put that state into a struct that's initialized and returned by a function in the package.

The non-explicit invocation of `init` functions means that you should document their behavior. For example, a package with an `init` function that loads files or accesses the network should call this out in a package-level comment so that security-conscious users of your code aren't surprised by unexpected I/O.

Working with Modules

You've seen how to work with packages within a single module, and now it's time to see how to integrate with third-party modules and the packages within them. After that, you'll learn about publishing and versioning our own modules and Go's centralized services: pkg.go.dev, the module proxy, and the checksum database.

Importing Third-Party Code

So far, you've imported packages from the standard library like `fmt`, `errors`, `os`, and `math`. Go uses the same import system to integrate

packages from third parties. Unlike many other compiled languages, Go always builds applications from source code into a single binary file. This includes the source code of your module and the source code of all the modules on which your module depends. Just as we saw when we imported a package from within our own module, when you import a third-party package, you specify the location in the source code repository where the package is located.

Let's look at an example. I mentioned back in Chapter 2 that you should never use floating point numbers when you need an exact representation of a decimal number. If you do need an exact representation, one good option is the `decimal` module from ShopSpring. We are also going to look at a simple formatting module that I've written for this book. Both of these modules are used in a small program that can be found in the GitHub organization for the book. This program accurately calculates the price of an item with the tax included and prints the output in a neat format.

The following code is in *main.go*:

```
package main

import (
    "fmt"
    "log"
    "os"

    "github.com/learning-go-book-2e/formatter"
    "github.com/shopspring/decimal"
)

func main() {
    if len(os.Args) < 3 {
        fmt.Println("Need two parameters: amount and percent")
        os.Exit(1)
    }
    amount, err := decimal.NewFromString(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    percent, err := decimal.NewFromString(os.Args[2])
    if err != nil {
        log.Fatal(err)
```

```
    }
    percent = percent.Div(decimal.NewFromInt(100))
    total := amount.Add(amount.Mul(percent)).Round(2)
    fmt.Println(formatter.Space(80, os.Args[1], os.Args[2],
                                total.StringFixed(2)))
}
```

The two imports `github.com/learning-go-book-2e/formatter` and `github.com/shopspring/decimal` specify third-party imports. Note that they include the location of the package in the repository. Once imported, we access the exported items in these packages just like any other imported package.

Before we build our application, look at the *go.mod* file. Its contents should be:

```
module github.com/learning-go-book-2e/money

go 1.19
```

If we try to do a build, we get the following message:

```
$ go build
main.go:8:2: no required module provides package
    github.com/learning-go-book-2e/formatter; to add it:
        go get github.com/learning-go-book-2e/formatter
main.go:9:2: no required module provides package
    github.com/shopspring/decimal; to add it:
        go get github.com/shopspring/decimal
```

As the errors indicate, you cannot build the program until we add references to the third-party modules to our `go.mod` file. The `go get` command downloads modules and updates the `go.mod` file. You have two options when using `go get`. The simplest option is to tell `go get` to scan your module's source code and add any modules that are found in `import` statements to `go.mod`:

```
$ go get ./...
```

```
go: downloading github.com/shopspring/decimal v1.3.1
go: downloading github.com/learning-go-book-2e/formatter v0.0.0-
20220918024742-1835a89362c9
go: downloading github.com/fatih/color v1.13.0
go: downloading github.com/mattn/go-colorable v0.1.9
go: downloading github.com/mattn/go-isatty v0.0.14
go: downloading golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c
go: added github.com/fatih/color v1.13.0
go: added github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-
1835a89362c9
go: added github.com/mattn/go-colorable v0.1.9
go: added github.com/mattn/go-isatty v0.0.14
go: added github.com/shopspring/decimal v1.3.1
go: added golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c
```

Because the location of the package is in the source code, `go get` is able to get the package's module and download it. If you look in the *go.mod* file now, you'll see:

```
module github.com/learning-go-book-2e/money

go 1.19

require (
    github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-
1835a89362c9
    github.com/shopspring/decimal v1.3.1
)

require (
    github.com/fatih/color v1.13.0 // indirect
    github.com/mattn/go-colorable v0.1.9 // indirect
    github.com/mattn/go-isatty v0.0.14 // indirect
    golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c // indirect
)
```

The first `require` section of the *go.mod* file lists the modules that you've imported into your module. After the module name is a version number. In the case of the `formatter` module, it doesn't have a version tag, so Go makes up a *pseudo-version*.

You also see a second `require` directive section that has modules with an *indirect* comment. One of these modules (github.com/fatih/color) is directly

used by `formatter`. It in turn depends on the other three modules in the second `require` directive section. All of the modules used by all of your module's dependencies (and your dependencies' dependencies, and so on) are included in the `go.mod` file for your module. The ones that are only used in dependencies are marked as indirect.

In addition to updating *go.mod*, a *go.sum* file is also created. For each module in the dependency tree of your project, the *go.sum* file has one or two entries: one with the module, its version, and a hash of the module, the other with the hash of the *go.mod* file for the module. Here's what our *go.sum* file looks like:

```
github.com/fatih/color v1.13.0 h1:8LOYc1KYPPmyKMuN8QV2DNRWNbLo6LZ0iLs...
github.com/fatih/color v1.13.0/go.mod h1:kLAiJbzzSOZDVNGyDpeOxJ47H46q...
github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-1835a8...
github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-1835a8...
github.com/mattn/go-colorable v0.1.9 h1:sqDoxXbdeALODt0DAeJCVp38ps9Zo...
github.com/mattn/go-colorable v0.1.9/go.mod h1:u6P/XSegPjTcexA+o6vUJr...
github.com/mattn/go-isatty v0.0.12/go.mod h1:cbi8OIDigv2wuxKPP5vlRcQ1...
github.com/mattn/go-isatty v0.0.14 h1:yVuAays6BHfxijgZPzw+3Zlu5yQgKGP...
github.com/mattn/go-isatty v0.0.14/go.mod h1:7GGIvUiUoEMVVmxf/4nioHXj...
github.com/shopspring/decimal v1.3.1 h1:2Usl1nmF/WZucqkFZhnfFYxxxu8LG...
github.com/shopspring/decimal v1.3.1/go.mod h1:DKyhrW/HYNuLGql+MJL6WC...
golang.org/x/sys v0.0.0-20200116001909-b77594299b42/go.mod h1:h1NjWce...
golang.org/x/sys v0.0.0-20200223170610-d5e6a3e2c0ae/go.mod h1:h1NjWce...
golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c h1:F1jZWGFhYfh0Ci...
golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c/go.mod h1:oPkhp1M...
```

You'll see what these hashes are used for in "Module Proxy Servers". You might also notice that there are multiple versions of some of the dependencies. We'll talk about that in "Minimal Version Selection".

Let's validate that our modules are now set up correctly. Run `go build` again, and then run the `money` binary and pass it some arguments:

```
$ go build
$ ./money 99.99 7.25
99.99          7.25                                          107.24
```

> **NOTE**
>
> Our sample program was checked in without *go.sum* and with an incomplete *go.mod*. This was done so you could see what happens when these files are populated. When committing your own modules to source control, always include up-to-date *go.mod* and *go.sum* files. Doing so specifies exactly what versions of your dependencies are being used. This enables *repeatable builds*; when anyone else (including your future self) builds this module, they will get the exact same binary.

As I mentioned, there's another way to use `go get`. Instead of telling it to scan your source code to discover modules, you can pass the module paths to `go get`. To see this work, roll back the changes to your `go.mod` file and remove the `go.sum` file. On Unix-like systems, the following commands will do this:

```
$ git restore go.mod
$ rm go.sum
```

Now, pass the module paths to `go get` directly:

```
$ go get github.com/learning-go-book-2e/formatter
go: added github.com/learning-go-book-2e/formatter v0.0.0-20200921021027-
5abc380940ae
$ go get github.com/shopspring/decimal
go: added github.com/shopspring/decimal v1.3.1
```

> **NOTE**
>
> Sharp-eyed readers might notice that when we used `go get` a second time, the `go: downloading` messages aren't display. The reason for this is that Go maintains a *module cache* on your local computer. Once a version of a module is downloaded, a copy is kept in the cache. Source code is pretty compact and drives are pretty large, so this isn't usually a concern. However, if you want to delete the module cache, use the command `go clean -modcache`.

Take a look at the contents of `go.mod` and they'll look a little different than before:

```
module github.com/learning-go-book-2e/money

go 1.19

require (
    github.com/fatih/color v1.13.0 // indirect
    github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-
1835a89362c9 // indirect
    github.com/mattn/go-colorable v0.1.9 // indirect
    github.com/mattn/go-isatty v0.0.14 // indirect
    github.com/shopspring/decimal v1.3.1 // indirect
    golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c // indirect
)
```

Notice that all of the imports are marked as *indirect*, not just the ones that came from `formatter`. When you run `go get` and pass it a module name, it doesn't check your source code to see if the module you specified is used within your main module. Just to be safe, it adds an indirect comment.

If you want to fix this label automatically, use the command `go mod tidy`. It scans your source code and synchronizes the `go.mod` and `go.sum` files with your module's source code, adding and removing module references. It also makes sure that the indirect comments are correct.

You might be wondering why you would want to bother using `go get` with a module name. The reason is that it allows you to update the version of an individual module.

Working with Versions

Let's see how Go's module system uses versions. I've written a simple module that we're going to use in another tax collection program. In *main.go*, we have the following third-party imports:

```
"github.com/learning-go-book-2e/simpletax"
"github.com/shopspring/decimal"
```

Like before, our sample program wasn't checked in with *go.mod* and *go.sum* updated, so we could see what happens. When we build our program, we see the following:

```
$ go get ./...
go: downloading github.com/learning-go-book-2e/simpletax v1.1.0
go: added github.com/learning-go-book-2e/simpletax v1.1.0
go: added github.com/shopspring/decimal v1.3.1
$ go build
```

The *go.mod* file has been updated to:

```
module github.com/learning-go-book-2e/region_tax

go 1.19

require (
    github.com/learning-go-book-2e/simpletax v1.1.0
    github.com/shopspring/decimal v1.3.1
)
```

We also have a *go.sum* with hashes for our dependencies. Let's run our code and see if it's working:

```
$ ./region_tax 99.99 12345
2022/09/19 22:04:38 unknown zip: 12345
```

That looks like an unexpected answer. There might be a bug in the latest version of the module. By default, Go picks the latest version of a dependency when you add it to your module. However, one of the things that makes versioning useful is that you can specify an earlier version of a module. First we can see what versions of the module are available with the `go list` command:

```
$ go list -m -versions github.com/learning-go-book-2e/simpletax
github.com/learning-go-book-2e/simpletax v1.0.0 v1.1.0
```

By default, the `go list` command lists the packages that are used in your module. The `-m` flag changes the output to list the modules instead, and the `-versions` flag changes `go list` to report on the available versions for the specified module. In this case, we see that there are two versions, v1.0.0 and v1.1.0. Let's downgrade to version v1.0.0 and see if that fixes our problem. We do that with the `go get` command:

```
$ go get github.com/learning-go-book-2e/simpletax@v1.0.0
go: downloading github.com/learning-go-book-2e/simpletax v1.0.0
go: downgraded github.com/learning-go-book-2e/simpletax v1.1.0 => v1.0.0
```

The `go get` command lets us work with modules, updating the versions of our dependencies.

Now if you look at *go.mod*, you'll see the version has been changed:

```
module github.com/learning-go-book-2e/region_tax

go 1.19

require (
    github.com/learning-go-book-2e/simpletax v1.0.0
    github.com/shopspring/decimal v1.3.1
)
```

We also see in *go.sum* that it contains both versions of `simpletax`:

```
github.com/learning-go-book-2e/simpletax v1.0.0 h1:KZU8aXRCHkvgFmBWkV...
github.com/learning-go-book-2e/simpletax v1.0.0/go.mod h1:lR4YYZwbDTI...
github.com/learning-go-book-2e/simpletax v1.1.0 h1:sG83gscauX/b8yKKY9...
github.com/learning-go-book-2e/simpletax v1.1.0/go.mod h1:lR4YYZwbDTI...
```

This is fine; if you change a module's version, or even remove a module from your module, there still might be an entry for it in *go.sum*. This doesn't cause any problems.

When we build and run our code again, the bug is fixed:

```
$ go build
$ ./region_tax 99.99 12345
107.99
```

SEMANTIC VERSIONING

Software has had version numbers from time immemorial, but there has been little consistency in what version numbers mean. The version numbers attached to Go modules follow the rules of *semantic versioning*, also known as *SemVer*. By requiring semantic versioning for modules, Go makes its module management code simpler while ensuring that users of a module understand what a new release promises.

If you aren't familiar with SemVer, check out the full specification. The very short explanation is that semantic versioning divides a version number into three parts: the *major* version, the *minor* version, and the *patch* version, which are written as `major.minor.patch` and preceded by a `v`. The patch version number is incremented when fixing a bug, the minor version number is incremented (and the patch version is set back to 0) when a new, backward-compatible feature is added, and the major version number is incremented (and minor and patch are set back to 0) when making a change that breaks backward compatibility.

Minimal Version Selection

At some point, your module will depend on two or more modules that all depend on the same module. As often happens, these modules declare that they depend on different minor or patch versions of that module. How does Go resolve this?

The module system uses the principle of *minimal version selection*. This means that you will always get the lowest version of a dependency that is declared to work in all of the *go.mod* files across all of your dependencies. Let's say that your module directly depends on modules A, B, and C. All

three of these modules depend on module D. The *go.mod* file for module A declares that it depends on v1.1.0, module B declares that it depends on v1.2.0, and module C declares that it depends on v1.2.3. Go will import module D only once, and it will choose version v1.2.3, as that, in the words of the Go Modules Reference, is the minimum version that satisfies all requirements.

We can see this in action with our sample program from "Importing Third-Party Code". The command `go mod graph` shows the dependency graph of your module and all of its dependencies. Here are a few lines of its output:

```
github.com/learning-go-book-2e/money github.com/fatih/color@v1.13.0
github.com/learning-go-book-2e/money github.com/mattn/go-colorable@v0.1.9
github.com/learning-go-book-2e/money github.com/mattn/go-isatty@v0.0.14
github.com/fatih/color@v1.13.0 github.com/mattn/go-colorable@v0.1.9
github.com/fatih/color@v1.13.0 github.com/mattn/go-isatty@v0.0.14
github.com/mattn/go-colorable@v0.1.9 github.com/mattn/go-isatty@v0.0.12
```

Each line lists two modules, the first being the parent and the second being the dependency and its version. You'll notice the `github.com/fatih/color` module is declared to depend on version v0.0.14 of `github.com/mattn/go-isatty`, while `github.com/mattn/go-colorable` depends on v0.0.12. The Go compiler selects version v0.0.14 to use, because it is the minimal version that meets all requirements. This is despite the fact that, as of this writing, the latest version of `github.com/mattn/go-isatty` is v0.0.16. Our minimal version requirement is met with v0.0.14, so that's what is used.

This system isn't perfect. You might find that while module A works with version v1.1.0 of module D, it does not work with version v1.2.3. What do you do then? Go's answer is that you need to contact the module authors to fix their incompatibilities. The *import compatibility rule* says "If an old package and a new package have the same import path, the new package must be backwards compatible with the old package." This means that all minor and patch versions of a module must be backward compatible. If they aren't, it's a bug. In our hypothetical example, either module D needs to be

fixed because it broke backward compatibility, or module A needs to be fixed because it made a faulty assumption about the behavior of module D.

You might not find this answer satisfying, but it's honest. Some build systems, like npm, will include multiple versions of the same package. This can introduce its own set of bugs, especially when there is package-level state. It also increases the size of your application. In the end, some things are better solved by community than code.

Updating to Compatible Versions

What about the case where you explicitly want to upgrade a dependency? Let's assume that after we wrote our initial program, there are three more versions of `simpletax`. The first fixes problems in the initial v1.1.0 release. Since it's a bug patch release with no new functionality, it would be released as v1.1.1. The second keeps the current functionality, but also adds a new function. It would get the version number v1.2.0. Finally, the third fixes a bug that was found in version v1.2.0. It has the version number v1.2.1.

To upgrade to the bug patch release for the current minor version, use the command `go get -u=patch github.com/learning-go-book-2e/simpletax`. Since we had downgraded to v1.0.0, we would remain on that version, since there is no patch version with the same minor version.

If we upgraded to version v1.1.0 using `go get github.com/learning-go-book-2e/simpletax@v1.1.0` and then ran `go get -u=patch github.com/learning-go-book-2e/simpletax`, we would be upgraded to version v1.1.1.

Finally, use the command `go get -u github.com/learning-go-book-2e/simpletax` to get the most recent version of `simpletax`. That upgrades us to version v1.2.1.

Updating to Incompatible Versions

Let's go back to our program. We're expanding to Canada, and luckily, there's a version of the `simpletax` module that handles both the US and Canada. However, this version has a slightly different API than the previous one, so its version is v2.0.0.

To handle incompatibility, Go modules follow the *semantic import versioning* rule. There are two parts to this rule:

- The major version of the module must be incremented.
- For all major versions besides 0 and 1, the path to the module must end in vN, where N is the major version.

The path changes because an import path uniquely identifies a package. By definition, incompatible versions of a package are not the same package. Using different paths means that you can import two incompatible versions of a package into different parts of your program, allowing you to upgrade gracefully.

Let's see how this changes our program. First, we are going to change our import of `simpletax` to:

```
"github.com/learning-go-book-2e/simpletax/v2"
```

This changes our import to refer to the v2 module.

Next, we're going to change the code in `main` to the following:

```
func main() {
    amount, err := decimal.NewFromString(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    zip := os.Args[2]
    country := os.Args[3]
    percent, err := simpletax.ForCountryPostalCode(country, zip)
    if err != nil {
        log.Fatal(err)
    }
    total := amount.Add(amount.Mul(percent)).Round(2)
```

```
    fmt.Println(total)
}
```

We are now reading a third parameter from the command line, which is the country code, and we call a different function in the `simpletax` package. When we call `go get ./...`, the dependency is automatically updated:

```
$ go get ./...
go: downloading github.com/learning-go-book-2e/simpletax/v2 v2.0.0
go: added github.com/learning-go-book-2e/simpletax/v2 v2.0.0
```

We can build and run the program and see our new output:

```
$ go build
$ ./region_tax 99.99 M4B1B4 CA
112.99
$ ./region_tax 99.99 12345 US
107.99
```

When you look at the *go.mod* file, you'll see that the new version of `simpletax` is included:

```
module github.com/learning-go-book-2e/region_tax

go 1.19

require (
    github.com/learning-go-book-2e/simpletax v1.0.0
    github.com/learning-go-book-2e/simpletax/v2 v2.0.0
    github.com/shopspring/decimal v1.3.1
)
```

And *go.sum* has been updated as well:

```
github.com/learning-go-book-2e/simpletax v1.0.0 h1:KZU8aXRCHkvgFmBWkV...
github.com/learning-go-book-2e/simpletax v1.0.0/go.mod h1:lR4YYZwbDTI...
github.com/learning-go-book-2e/simpletax v1.1.0 h1:sG83gscauX/b8yKKY9...
github.com/learning-go-book-2e/simpletax v1.1.0/go.mod h1:lR4YYZwbDTI...
github.com/learning-go-book-2e/simpletax/v2 v2.0.0 h1:EUFWy1BBA2omgkm...
```

```
github.com/learning-go-book-2e/simpletax/v2 v2.0.0/go.mod h1:yGLh6ngH...
github.com/shopspring/decimal v1.3.1 h1:2Usl1nmF/WZucqkFZhnfFYxxxu8LG...
github.com/shopspring/decimal v1.3.1/go.mod h1:DKyhrW/HYNuLGql+MJL6WC...
```

The old versions of `simpletax` are still referenced, even though they are no longer used. We can use `go mod tidy` to remove those unused versions. Then you'll only see v2.0.0 of simpletax referenced in *go.mod* and *go.sum*.

Vendoring

To ensure that a module always builds with identical dependencies, some organizations like to keep copies of their dependencies inside their module. This is known as *vendoring*. It's enabled by running the command `go mod vendor`. This creates a directory called *vendor* at the top level of your module that contains all of your module's dependencies. These dependencies are used in place of the module cache stored on your computer.

If new dependencies are added to *go.mod* or versions of existing dependencies are upgraded with `go get`, you need to run `go mod vendor` again to update the *vendor* directory. If you forget to do this, `go build`, `go run`, and `go test` will display an error message and refuse to run.

Older Go dependency management systems required vendoring, but with the advent of Go modules and proxy servers (see "Module Proxy Servers" for details), the practice is falling out of favor. One reason why you might still want to vendor is that it can make building your code faster and more efficient when working with some CI/CD (continuous integration/continuous delivery) pipelines. If a pipeline's build servers are ephemeral, the module cache may not be preserved. Vendoring dependencies allows these pipelines to avoid making multiple network calls to download dependencies every time a build is triggered. The downside is that it dramatically increases the size of your codebase in version control.

pkg.go.dev

While there isn't a single centralized repository of Go modules, there is a single service that gathers together documentation on Go modules. The Go team has created a site called *pkg.go.dev* that automatically indexes open source Go modules. For each module, the package index publishes the godocs, the license used, the *README*, the module's dependencies, and what open source modules depends on the module. You can see the info that *pkg.go.dev* has on our `simpletax` module in Figure 4-2.

github.com/learning-go-book/simpletax

Package simpletax v1.1.0 Latest

The latest major version is v2.

Published: 1 day ago | License: MIT | Module: github.com/learning-go-book/simpletax

Doc Overview Subdirectories **Versions** Imports Imported By Licenses

Versions in this module

v1 - github.com/learning-go-book/simpletax

v1.1.0 - 1 day ago

v1.0.0 - 1 day ago

Other modules containing this package

v2 - github.com/learning-go-book/simpletax/v2

v2.0.0 - 1 day ago

Figure 4-2. Use pkg.go.dev to find and learn about third-party modules

Publishing Your Module

Making your module available to other people is as simple as putting it in a version control system. This is true whether you are releasing your module as open source on a public version control system like GitHub or a private one that's hosted within a company. Since Go programs build from source code and use a repository path to identify themselves, there's no need to explicitly upload your module to a central library repository, like you do for Maven Central or npm. Make sure you check in both your *go.mod* file and your *go.sum* file.

When releasing an open source module, you should include a file named *LICENSE* in the root of your repository that specifies the open source license under which you are releasing your code. It's FOSS has a good resource for learning more about the various kinds of open source licenses.

Roughly speaking, you can divide open source licenses into two categories: permissive (which allows users of your code to keep their code private) and nonpermissive (which requires users of your code to make their code open source). While the license you choose is up to you, the Go community favors permissive licenses, such as BSD, MIT, and Apache. Since Go compiles third-party code directly into every application, the use of a nonpermissive license like the GPL would require people who use your code to release their code as open source as well. For many organizations, this is not acceptable.

One final note: do not write your own license. Few people will trust that it has been properly reviewed by a lawyer, and they can't tell what claims you are making on their module.

Versioning Your Module

Whether your module is public or private, you must properly version your module so that it works correctly with Go's module system. As long as you are adding functionality or patching bugs, the process is simple. Store your changes in your source code repository, then apply a tag that follows the semantic versioning rules we discussed in "Semantic Versioning".

Go's semantic versioning supports the concept of *pre-releases*. Let's assume that the current version of your module is tagged `v1.3.4`. You are working on version 1.4.0, it's not quite done, but you want to try importing it into another module. What you should do is append a hyphen (-) to the end of your version tag, followed by an identifer for the pre-release build. In this case, we'd use a tag like `v1.4.0-beta1` to indicate beta 1 of version 1.4.0 or `v1.4.0-rc2` to indicate release candidate 2. If you want to depend on a pre-release candidate, you must specify its version explicitly in `go get`, as Go will not automatically select a pre-release version.

If you reach a point where you need to break backward compatibility, the process is more complicated. As we saw when we imported version 2 of the `simpletax` module, a backward-breaking change requires a different import path. There are a few steps to take.

First you need to choose a way to store your new version. Go supports two ways for creating the different import paths:

- Create a subdirectory within your module named *vN*, where *N* is the major version of your module. For example, if you are creating version 2 of your module, call this directory *v2*. Copy your code into this subdirectory, including the *README* and *LICENSE* files.
- Create a branch in your version control system. You can put either the old code or the new code on the new branch. Name the branch *vN* if you are putting the new code on the branch, or *vN-1* if you are putting the old code there. For example, if you are creating version 2 of your module and want to put version 1 code on the branch, name the branch *v1*.

After you decide how to store your new code, you need to change the import path in the code in your subdirectory or branch. The module path in your *go.mod* file must end with */vN*, and all of the imports within your module must use */vN* as well. Going through all of your code can be tedious, but Marwan Sulaiman has created a tool that automates the work. Once the paths are fixed, go ahead and implement your changes.

> **NOTE**
>
> Technically, you could just change *go.mod* and your import statements, tag your main branch with the latest version, and not bother with a subdirectory or versioned branch. However, this is not a good practice. It will break Go code built with older versions of the language and makes it unclear where to find older major versions of your module.

When you are ready to publish your new code, place a tag on your repository that looks like vN.0.0. If you are using the subdirectory system or keeping the latest code on your main branch, tag the main branch. If you are placing your new code on a different branch, tag that branch instead.

You can find more details on updating your code to an incompatible version in the post Go Modules: v2 and Beyond on the Go Blog.

Overriding dependencies

Forks happen. While there's a bias against forking in the open source community, sometimes a module stops being maintained or you want to experiment with changes that aren't accepted by the module's owner. A `replace` directive redirects all references to a module across all of your module's dependencies and replaces them with the specified fork of the module. It looks like this:

```
replace github.com/jonbodner/proteus => github.com/someone_else/my_proteus
v1.0.0
```

The original module location is specified on the left side of the ⇒ and the replacement on the right. The right side must have a version specified, but

specifying a version is optional for the left side. If the version is specified, only that specific version will be replaced. If the version is not specified, any version of the original module will be replaced with the specific version of the fork.

A `replace` directive can also refer to path on your local file system:

```
replace github.com/jonbodner/proteus => ../projects/proteus
```

With a local `replace` directive, the module version is optional on both the left and right side.

> **WARNING**
>
> Avoid using local `replace` directives. They provided a way to modify multiple modules simultaneously before the invention of Go workspaces, but now they are a potential source of broken modules. (We will cover workspaces shortly.) If you share your module via version control, a module with local references in `replace` directives will probably not build for anyone else, since you cannot guarantee that other people will have the replacement modules in the same locations on their drive.

You also might want to block a specific version of a module from being used. Perhaps it has a bug or is incompatible with your module. Go provides the `exclude` directive to prevent a specific version of a module from being used:

```
exclude github.com/jonbodner/proteus v0.10.1
```

When a module version is excluded, any mentions of that module version in any dependent module are ignored. If a module version is excluded and it's the only version of that module that's required in your module's dependencies, use `go get` to add an indirect import of a different version of the module to your module's `go.mod` file so that your module still compiles.

Retracting a version of your module

Sooner or later, you will accidentally publish a version of your module that you don't want anyone to use. Perhaps it was released by accident before testing was complete. Maybe after its release, a critical vulnerability is discovered and no one should use it any more. No matter the reason, Go provides a way for you to indicate that certain versions of a module should ignored. This is done by adding a `retract` directive to the `go.mod` file of your module. It consists of the word `retract` and the semantic version that should no longer be used. If there is a range of versions that shouldn't be used, you can exclude all versions in that range by placing the upper and lower bounds within brackets, separated by a comma While it's not required, you should include a comment after a version or version range to explain the reason for the retraction.

If there are multiple non-sequential versions that you wish to retract, you can specify them with multiple `retract` directives. In the examples shown, version 1.50 is excluded, as are all versions from 1.7.0 to 1.8.5, inclusive.

```
retract v1.5.0 // not fully tested
retract [v1.7.0, v.1.8.5] // posts your cat photos to LinkedIn w/o permission
```

Adding a `retract` directive to `go.mod` requires you to create a new version of your module. If the new version only contains the retraction, you should retract it as well.

When a version is retracted, existing builds that specified the version will continue to work, but `go get` and `go mod tidy` will not upgrade to them. They will no longer appear as options when you use the `go list` command. If the most recent version of a module is retracted, it will no longer be matched with `@latest`; the highest unretracted version will match instead.

NOTE

While `retract` can be confused with `exclude`, there's a very important difference. You use `retract` to prevent others from using specific versions of *your* module. An `exclude` directive blocks you from using versions of another module.

Using Workspaces to modify modules simultaneously

There's one drawback to using your source code repository and its tags as a way to track your dependencies and their versions. If you want to make changes to two (or more) modules simultaneously, and you want to experiment with those changes across modules, you need a way for a local copy of a module to override the version of a module in the source code repository.

> **WARNING**
>
> You can find obsolete advice online to try to solve this issue with temporary `replace` directives in `go.mod` that point to local directories . Do not do this! It's too easy to forget to undo these changes before committing and pushing your code. Workspaces were introduced to avoid this anti-pattern.

Go uses *workspaces* to address this issue. A workspace allows you to have multiple modules downloaded to your computer and references between those modules will automatically resolve to the local source code instead of the code hosted in your repository.

> **NOTE**
>
> This section assumes that you have a GitHub account. If you don't, you can still follow along. I'm going to use the organization name `learning-go-book-2e`, but you should replace it with your GitHub account name or organization.

Let's start with two sample modules. Create a directory called `my_workspace` and inside that directory, create two more directories, `workspace_lib` and `workspace_app`. In the `workspace_lib` directory, run `go mod init github.com/learning-go-book-2e/workspace_lib` . Create a file called `lib.go` with the following content:

```
package workspace_lib
```

```
func AddNums(a, b int) int {
    return a + b
}
```

In the `workspace_app` directory, run `go mod init github.com/learning-go-book-2e/workspace_app`. Create a file called `app.go` with the following contents:

```
package main

import (
    "fmt"
    "github.com/learning-go-book-2e/workspace_lib"
)

func main() {
    fmt.Println(workspace_lib.AddNums(2, 3))
}
```

In previous sections, we have used `go get ./...` to add `require` directives to `go.mod` . Let's see what happens if we try it here:

```
$ go get ./...
github.com/learning-go-book-2e/workspace_app imports
    github.com/learning-go-book-2e/workspace_lib: cannot find module
        providing package github.com/learning-go-book-2e/workspace_lib
```

Since `workspace_lib` hasn't been pushed to GitHub yet, we can't pull it in. If we try to run `go build`, we will get a similar error:

```
$ go build
app.go:5:2: no required module provides
    package github.com/learning-go-book-2e/workspace_lib; to add it:
        go get github.com/learning-go-book-2e/workspace_lib
```

Let's take advantage of workspaces to allow `workplace_app` to see the local copy of `workspace_lib`. Go to the `my_workspace` directory and run the following commands:

```
$ go work init ./workspace_app
$ go work use ./workspace_lib
```

This creates a `go.work` file in `my_workspace` with the following contents:

```
go 1.19

use (
    ./workspace_app
    ./workspace_lib
)
```

> **WARNING**
>
> The `go.work` file is meant for your local computer only. Do not commit it to source control!

Now if you build `workspace_app`, everything works:

```
$ cd workspace_app
$ go build
$ ./workspace_app
5
```

Now that we are sure that `workspace_lib` does the right thing, it can be pushed to GitHub. In GitHub, create an empty public repository called `workspace_lib` and then run the following commands in the `workspace_lib` directory:

```
$ git init
$ git add .
$ git commit -m "first commit"
$ git remote add origin git@github.com:learning-go-book-
2e/workspace_lib.git
$ git branch -M main
$ git push -u origin main
```

After running these commands, go to *https://github.com/learning-go-book-2e/workspace_lib/releases/new* (replacing `learning-go-book-2e` with your account or organization), and create a new release with the tag v0.1.0.

Now if we run `go get ./...`, the `require` directive is added, because there is a public module that can be downloaded:

```
$ go get ./...
go: downloading github.com/learning-go-book-2e/workspace_lib v0.1.0
go: added github.com/learning-go-book-2e/workspace_lib v0.1.0
$ cat go.mod
module github.com/learning-go-book-2e/workspace_app

go 1.19

require github.com/learning-go-book-2e/workspace_lib v0.1.0
```

Even though we now have a `require` directive referring to the public module, we can continue to make updates in our local workspace and they will be used instead. In `workspace_lib`, modify the `lib.go` file and add the following function:

```
func SubNums(a, b int) int {
    return a - b
}
```

In `workspace_app`, modify the `app.go` file and add the following line to the end of the `main` function:

```
    fmt.Println(workspace_lib.SubNums(2,3))
```

Now run `go build` and see it use the local module instead of the public one:

```
$ go build
$ ./workspace_app
5
-1
```

Once you have made the edits and you want to release your software, you need to update the version information in your modules' `go.mod` files to refer to the updated code. This requires you to do things in a specific order:

- commit your modules back to your source code repository in dependency order
- update the version tags on your modules in your source code repository
- use `go get` to update the version specified in `go.mod` in the modules that depend on the committed module
- repeat until all modified modules are committed.

If you have to make changes to `workspace_lib` in the future and want to test them in `workspace_app` without pushing back to GitHub and creating lots of temporary versions, you can `git pull` the latest versions of the modules into your workspace again and make your updates.

Module Proxy Servers

Rather than relying on a single, central repository for libraries, Go uses a hybrid model. Every Go module is stored in a source code repository, like GitHub or GitLab. But by default, `go get` doesn't fetch code directly from source code repositories. Instead, it sends requests to a *proxy server* run by Google. When the proxy server receives the `go get` request, it checks its cache to see if there has been a requrest for this version of this module before. If so, it returns the cached information. If a module or a version of a module isn't cached on the proxy server, it downloads the module from the module's repository, stores a copy, and returns the module. This allows the proxy server to keeps copies of every version of virtually all public Go modules.

In addition to the proxy server, Google also maintains a *checksum database*. It stores information on every version of every module cached by the proxy server. Just as the proxy server protects you from a module or a version of a

module being removed from the internet, the checksum database protects you against modifications to a version of a module. This could be malicious (someone has hijacked a module and slipped in malicious code), or it could be inadvertent (a module maintainer fixes a bug or adds a new feature and reuses an existing version tag). In either case, you don't want to use a module version that has changed because you won't be building the same binary and don't know what the effects are on your application.

Every time you download a module via `go get` or `go mod tidy`, the Go tools calculate a hash for the module and contact the checksum database to compare the calculated hash to the hash stored for that module's version. If they don't match, the module isn't installed.

Specifying a Proxy Server

Some people object to sending requests for third-party libraries to Google. There are a few options:

- You can disable proxying entirely by setting the `GOPROXY` environment variable to `direct`. You'll download modules directly from their repositories, but if you depend on a version that's removed from the repository, you won't be able to access it.
- You can run your own proxy server. Both Artifactory and Sonatype have Go proxy server support built into their enterprise repository products. The Athens Project provides an open source proxy server. Install one of these products on your network and then point `GOPROXY` to the URL.

Private Repositories

Most organizations keep their code in private repositories. If you want to use a private module in another Go module, you can't request it from Google's proxy server. Go will fall back to checking the private repository directly, but you might not want to leak the names of private servers and repositories to external services.

If you are using your own proxy server, or if you have disabled proxying, this isn't an issue. Running a private proxy server has some additional benefits. First, it speeds up downloading of third-party modules, as they are cached in your company's network. If accessing your private repositories requires authentication, using a private proxy server means that you don't have to worry about exposing authentication information in your CI/CD pipeline. The private proxy server is configured to authenticate to your private repositories (see the authentication configuration documentation for Athens), while the calls to the private proxy server are unauthenticated.

If you are using a public proxy server, you can set the `GOPRIVATE` environment variable to a comma-separated list of your private repositories. For example, if you set `GOPRIVATE` to:

```
GOPRIVATE=*.example.com,company.com/repo
```

Any module stored in a repository that's located at any subdomain of *example.com* or at a URL that starts with *company.com/repo* will be downloaded directly.

Additional Details

The Go Team has a complete Go Modules Reference available online. In addition to what is covered in this chapter, the Module Reference also covers topics like using version control systems besides `git`, the structure of the module cache, additional environment variables for controlling module lookup behavior, and the REST API for the checksum database.

Exercises

1. Create a module in your own public repository. This module has a single function named `Add` with two `int` parameters and one `int` return value. This function adds the two parameters together and returns them. Make this version v1.0.0.

2. Add godoc comments to your module that describe the package and the `Add` function. Be sure to include a link to *https://www.mathsisfun.com/numbers/addition.xhtml* in your Add function godoc. Make this version v1.0.1.

3. Change `Add` to make it generic. Import the `golang.org/x/exp/constraints` package. Combine the `Integer` and `Float` types in that package to create an interface called `Number`. Rewrite `Add` to take in two parameters of type `Number` and return a value of type `Number`. Version your module again. Because this is a backwards-breaking change, this should be v2.0.0 of your module.

Wrapping Up

In this chapter, you've learned how to organize code and interact with the ecosystem of Go source code. You've seen how modules work, how to organize your code into packages, how to use third-party modules, and how to release modules of your own. In the next chapter, you're going to take a look at more of the development tools that are included with Go, learn about some essential third party tools, and explore some techniques to give you better control over your build process.

Chapter 5. Go Tooling

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

A programming language doesn't exist in isolation. In order to be useful, there must also be tools that help the developer turn source code into an executable. Since Go is intended to address the problems that software engineers face today and to help them build quality software, careful thought has been put into tooling that simplifies tasks that are often difficult with other development platforms. This includes improvements in how you build code, format it, update it, validate it, distribute your code, and even how your users install your code.

We have already covered many of the bundled Go tools: `go vet`, `go fmt`, `go mod`, `go get`, `go list`, `go work`, `go doc`, and `go build`. The testing support provided by the `go test` tool is so extensive, it is covered by itself in Chapter 15. In this chapter, we will explore additional tools that make Go development great, both from the Go team and from third parties.

Use go run to try out small programs

Go is a compiled language, which means that before Go code is run, it must be converted into a executable file. This is in contrast to interpreted languages like Python or JavaScript, where you are able to write a quick script to test an idea and execute it immediately. Having that rapid feedback cycle is important, so Go provides similar functionality via the `go run` command. It builds and executes a program in one step. Let's go back to our first program from Chapter 1. Put it in a file called *hello.go*:

```
package main

import "fmt"

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

(You can also find this code in the repo for chapter 11 at `sample_code/gorun`)

Once the file is saved, use the `go run` command to build and execute it:

```
go run hello.go
Hello, world!
```

If you look inside the directory after running the `go run` command, you see that no binary has been saved there; the only file in the directory is the *hello.go* file we just created. Where did the executable (no pun intended) go?

The `go run` command does in fact compile your code into a binary. However, the binary is built in a temporary directory. The `go run` command builds the binary, executes the binary from that temporary directory, and then deletes the binary after your program finishes. This makes the `go run` command useful for testing out small programs or using Go like a scripting language.

> **TIP**
>
> Use `go run` when you want to treat a Go program like a script and run the source code immediately.

Add Third-Party Tools with go install

While some people choose to distribute their Go programs as pre-compiled binaries, tools written in Go can also be built from source and installed on your computer via the `go install` command.

As we saw in "Publishing Your Module", Go modules are identified via their source code repositories. The `go install` command takes an argument, which is the path to the main package in a module's source code repository, followed by an `@` and the version of the tool you want (if you just want to get the latest version, use `@latest`). It then downloads, compiles, and installs the tool.

By default, `go install` places binaries into the *go/bin* directory within your home directory. Set the *GOBIN* environment variable to change this location. It is strongly recommended that you add the `go install` directory to your executable search path (this is done by modifying the *PATH* environment variable on both Unix and Windows). For simplicity, all of the examples assume that you've added this directory.

> **NOTE**
>
> There are other environment variables that are recognized by the `go` tool. You can get a complete list, along with a brief description of each variable, using the `go help environment` command. Many of them control low-level behavior that can be safely ignored. We'll point out the relevant ones as needed.
>
> Some online resources tell you to set the GOROOT or GOPATH environment variables. GOROOT specifies the location where your Go development environment is installed and GOPATH was used to store all Go source code, both your own and third-party dependencies. Setting these variables is no longer necessary; the `go` tool figures out GOROOT automatically and GOPATH-based development has been superseded by modules.

Let's look at a quick example. Jaana Dogan created a great Go tool called `hey` that load tests HTTP servers. You can point it at the website of your choosing or an application that you've written. Here's how to install `hey` with the `go install` command:

```
$ go install github.com/rakyll/hey@latest
go: downloading github.com/rakyll/hey v0.1.4
go: downloading golang.org/x/net v0.0.0-20181017193950-04a2e542c03f
go: downloading golang.org/x/text v0.3.0
```

This downloads `hey` and all of its dependencies, builds the program, and installs the binary.

> **NOTE**
>
> As we covered in "Module Proxy Servers", the contents of Go repositories are cached in proxy servers. Depending on the repository and the values in your `GOPROXY` environment variable, `go install` may download from a proxy or directly from a repository. If `go install` downloads directly from a repository, it relies on command-line tools being installed on your computer. For example, you must have Git installed to download from GitHub.

Now that we have built and installed *hey*, we can run it with:

```
$ hey https://go.dev

Summary:
  Total:          2.1272 secs
  Slowest:        1.4227 secs
  Fastest:        0.0573 secs
  Average:        0.3467 secs
  Requests/sec:   94.0181
```

If you have already installed a tool and want to update it to a newer version, rerun `go install` with the newer version specified or with `@latest`:

```
$ go install github.com/rakyll/hey@latest
```

Of course, you don't need to leave programs installed via `go install` in the `go/bin` directory; they are regular executable binaries and can be stored anywhere on your computer. Likewise, you don't have to distribute programs written in Go using `go install`; you can put a binary up for download. However, `go install` is convenient, and it has become the method of choice for distributing third-party developer tools. Here are some of the most popular.

Improve import formatting with goimports

There's an enhanced version of `go fmt` available called `goimports` that also cleans up your import statements. It puts them in alphabetical order, removes unused imports, and attempts to guess any unspecified imports. Its guesses are sometimes inaccurate, so you should insert imports yourself.

You can download `goimports` with the command `go install golang.org/x/tools/cmd/goimports@latest`. You run it across your project with the command:

```
$ goimports -l -w .
```

The `-l` flag tells `goimports` to print the files with incorrect formatting to the console. The `-w` flag tells `goimports` to modify the files in-place. The `.` specifies the files to be scanned: everything in the current directory and all of its subdirectories.

Code-quality scanners

Back in "go vet", we looked at the built-in tool `go vet`, which scans source code for common programming errors. There are many third-party tools to check code style and scan for potential bugs that are missed by `go vet`. These tools are often called *linters*. (The term "linter" comes from the Unix team at Bell Labs; the first linter was written in 1978.) In addition to likely programming errors, some of the changes suggested by these tools include properly naming variables, formatting error messages, and placing comments on public methods and types. These aren't errors since they don't keep your programs from compiling or make your program run incorrectly, but they do flag situations where you are writing non-idiomatic code.

When you add linters to your build process, follow the old maxim of "trust, but verify". Because the kinds of issues that linters find are more fuzzy, they sometimes have false positives and false negatives. This means that you don't *have* to make the changes that they suggest, but you should take the suggestions seriously. Go developers expect code to look a certain way and follow certain rules, and if your code does not, it sticks out.

If you find a linter's suggestion to be unhelpful, each linting tool allows you to add a comment to your source code that blocks the errant result (the format of the comment varies from linter to linter; check each tool's documentation to learn what to write). Your comment should also include an explanation of why you are ignoring the linter's finding, so code reviewers (and future you, when you look back on your source code in six months) understand your reasoning.

staticcheck

If you had to pick one third-party scanner, use `staticcheck`. It is supported by many companies that are active in the Go community, includes more than 150 code quality checks, and tries to produce few to no false positives. It is installed via `go install honnef.co/go/tools/cmd/staticcheck@latest`. Invoke it with the command `staticcheck ./...` to examine your module.

Here's something that `staticcheck` finds that `go vet` does not:

```
package main

import "fmt"

func main() {
    s := fmt.Sprintf("Hello")
    fmt.Println(s)
}
```

(You can also find this code in the repo for chapter 11 at `sample_code/staticcheck_test`)

If you run `go vet` on this code, it doesn't find anything wrong. However, `staticcheck` notices a problem:

```
$ staticcheck ./...
main.go:6:7: unnecessary use of fmt.Sprintf (S1039)
```

Pass the code in parenthesis to `staticcheck` with the `-explain` flag for an explanation of the issue:

```
$ staticcheck -explain S1039
Unnecessary use of fmt.Sprint

Calling fmt.Sprint with a single string argument is unnecessary
and identical to using the string directly.

Available since
    2020.1
```

```
Online documentation
    https://staticcheck.io/docs/checks#S1039
```

revive

Another good linting option is `revive`. It is based on `golint`, a tool that used to be maintained by the Go team. Install `revive` with the command `go install github.com/mgechev/revive@latest` . By default, it only enables the rules that were present in `golint`. It can find style and code-quality issues like exported identifiers that don't have comments, variables that don't follow naming conventions, or error return values that aren't last.

With a configuration file, you can turn on many more rules. For example, to enable a check for shadowing of universe block identifiers, create a file named `built_in.toml` with the following contents:

```
[rule.redefines-builtin-id]
```

If you scan the following code:

```
package main

import "fmt"

func main() {
    true := false
    fmt.Println(true)
}
```

(You can also find this code in the repo for chapter 11 at `sample_code/revive_test`)

You'll get this warning:

```
$ revive -config built_in.toml ./...
main.go:6:2: assignment creates a shadow of built-in identifier true
```

Other rules that can be enabled are focused on opinionated code organization, like limiting the number of lines in a function or number of public structs in a file. There are even rules for evaluating the complexity of the logic in a function. Documentation on configuring `revive` is at *https://revive.run/docs* and the supported rules are described at *https://revive.run/r* .

golangci-lint

Finally, if you'd rather take the buffet approach to tool selection, there's `golangci-lint`. It is designed to make it as efficient as possible to configure and run over 50 code quality tools, including `go vet`, `staticcheck`, and `revive`.

While you can use `go install` to install `golangci-lint`, it is recommended that you download a binary version instead. Follow the installation instructions on the website. Once it is installed, you run `golangci-lint` with the command:

```
$ golangci-lint run
```

Back in "Unused Variables", we looked at a program with variables that were set to values that were never read and mentioned that `go vet` and the go compiler were unable to detect these issues. Neither `staticcheck` nor `revive` catches this problem. However, one of the tools bundled with `golangci-lint` does:

```
$ golangci-lint run
main.go:6:2: ineffectual assignment to x (ineffassign)
    x := 10
    ^
main.go:9:2: ineffectual assignment to x (ineffassign)
    x = 30
    ^
```

We can also use `golangci-lint` to provide shadowing checks that go beyond what `revive` can do. Configure `golangci-lint` to detect shadowing of both universe block identifiers and identifiers within your own code by putting the following configuration into a file named `.golangci.yml` in the directory where you run golangci-lint:

```
linters:
  enable:
    - govet
    - predeclared

linters-settings:
  govet:
    check-shadowing: true
    settings:
      shadow:
        strict: true
    enable-all: true
```

With these settings, running `golangci-lint` on this code:

```
package main

import "fmt"

var b = 20

func main() {
        true := false
        a := 10
        b := 30
        if true {
                a := 20
                fmt.Println(a)
        }
        fmt.Println(a, b)
}
```

detects the following issues:

```
$ golangci-lint run
main.go:5:5: var `b` is unused (unused)
```

```
var b = 20
    ^
main.go:10:2: shadow: declaration of "b" shadows declaration at line 5 (govet)
    b := 30
    ^
main.go:12:3: shadow: declaration of "a" shadows declaration at line 9 (govet)
        a := 20
        ^
main.go:8:2: variable true has same name as predeclared identifier
(predeclared)
    true := false
    ^
```

(You can find both golangci-lint code samples in the repo for chapter 11 at `sample_code/golangci-lint_test`)

Because `golangci-lint` runs so many tools (as of this writing, it runs 7 different tools by default and allows you to enable more than 50 more), it's inevitable that your team may disagree with some of its suggestions. Review the documentation at *https://golangci-lint.run/usage/linters/* to understand what each tool can do. Once you come to agreement on which linters to enable, update the *.golangci.yml* at the root of your module and commit it to source control. Check out the documentation for the file format.

> **NOTE**
>
> While `golangci-lint` allows you to have a configuration file in your home directory, don't put one there if you are working with other developers. Unless you enjoy adding hours of silly arguments to your code reviews, you want to make sure that everyone is using the same code quality tests and formatting rules.

I recommend that you start off using `go vet` as a required part of your automated build process. Add `staticcheck` next since it produces few false positives. When you are interested in configuring tools and setting code quality standards, look at `revive`, but be aware that it might have false positives and false negatives, so you can't require your team to fix every

issue it reports. Once you are used to their recommendations, try out `golangci-lint` and tweak its settings until it works for your team.

Use govulncheck to scan for vulnerable dependencies

There's one kind of code quality that isn't enforced by the tools we've looked at so far: software vulnerabilities. Having a rich ecosystem of third-party modules is fantastic, but clever hackers find a security vulnerabilities in libraries and exploit them. Developers patch these bugs when they are reported, but how do we ensure that the software that uses a vulnerable version of a library library is updated to the fixed version?

The Go team has released a tool called `govulncheck` to address this situation. It scans through your dependencies and finds known vulnerabilities in both the standard library and in third party libraries imported into your module. These vulnerabilities are reported in a public database maintained by the Go team. You can install it with:

```
$ go install golang.org/x/vuln/cmd/govulncheck@latest
```

Let's take a look at a small program to see the vulnerability checker in action. Go to *https://github.com/learning-go-book-2e/vulnerable* and download the repository. The source code in `main.go` is very simple. It imports a third-party YAML library and uses it to load a small YAML string into a struct:

```
func main() {
    info := Info{}

    err := yaml.Unmarshal([]byte(data), &info)
    if err != nil {
        fmt.Printf("error: %v\n", err)
    }
    fmt.Printf("%+v\n", info)
}
```

The `go.mod` file contains the required modules and their versions:

```
module github.com/learning-go-book-2e/vulnerable

go 1.19

require gopkg.in/yaml.v2 v2.2.7

require gopkg.in/check.v1 v1.0.0-20201130134442-10cb98267c6c // indirect
```

Let's see what happens when you run `govulncheck` on this project:

```
$ govulncheck ./...
govulncheck is an experimental tool.
  Share feedback at https://go.dev/s/govulncheck-feedback.

Scanning for dependencies with known vulnerabilities...
Found 1 known vulnerability.

Vulnerability #1: GO-2020-0036
  Due to unbounded aliasing, a crafted YAML file can cause
  consumption of significant system resources. If parsing user
  supplied input, this may be used as a denial of service vector.

  Call stacks in your code:
      main.go:25:23: github.com/learning-go-book-2e/vulnerable.main
        calls gopkg.in/yaml.v2.Unmarshal

  Found in: gopkg.in/yaml.v2@v2.2.7
  Fixed in: gopkg.in/yaml.v2@v2.2.8
  More info: https://pkg.go.dev/vuln/GO-2020-0036
```

This module is using an old and vulnerable version of the YAML package. `govulncheck` helpfully gives the exact line in the codebase that calls the problematic code.

> **NOTE**
>
> If `govulncheck` knows there a vulnerability in a module that your code uses, but can't find an explicit call to the buggy part of the module, you'll get a less severe warning. The message informs you of the library's vulnerability and what version resolves the issue, but it will also let you know that your module is likely not affected.

Let's update to a fixed version and see if that solves our problem:

```
$ go get -u=patch gopkg.in/yaml.v2
go: downloading gopkg.in/yaml.v2 v2.2.8
go: upgraded gopkg.in/yaml.v2 v2.2.7 => v2.2.8
$ govulncheck ./...
govulncheck is an experimental tool.
  Share feedback at https://go.dev/s/govulncheck-feedback.

Scanning for dependencies with known vulnerabilities...
No vulnerabilities found.
```

Remember, you should always strive for the smallest possible change to your project's dependencies since that makes it less likely that a change in a dependency breaks your code. For that reason, we update to the most recent patch version for v2.2.x, which is v2.2.8. When `govulncheck` is run again, there are no known issues.

While `govulncheck` currently requires a `go install` to download it, it's likely that it will be added to the standard toolset eventually. In the meantime, be sure to install and run it against your projects as a regular part of your builds. You can learn more about it in the blog post that announced it.

Embedding content into your program

Many programs are distributed with directories of support files; you might have web page templates or some standard data that's loaded when a program starts. If a Go program needs support files, you could include a

directory of files, but this takes away one of the advantages of Go; its ability to compile to a single binary that's easy to ship and distribute. However, there's another option. You can embed the contents of the files within your Go binary using `go:embed` comments.

You can find a program demonstrating embedding on Github in the `sample_code/embed_passwords` directory in the chapter 11 repo. It checks to see if a password is one of the 10,000 most commonly used passwords. Rather than write that list of passwords directly into the source code, we're going to embed it.

The code in main.go is straightforward:

```
package main

import (
    _ "embed"
    "fmt"
    "os"
    "strings"
)

//go:embed passwords.txt
var passwords string

func main() {
    pwds := strings.Split(passwords, "\n")
    if len(os.Args) > 1 {
        for _, v := range pwds {
            if v == os.Args[1] {
                fmt.Println("true")
                os.Exit(0)
            }
        }
        fmt.Println("false")
    }
}
```

You must do two things to enable embedding. First, the `embed` package must be imported. The Go compiler uses this import as a flag to indicate that embedding should be enabled. Because this sample code isn't referring to anything exported from the `embed` package, we use a blank import, which

was discussed in "The init Function: Avoid if Possible". The only symbol exported from `embed` is `FS`. We'll use it in our next example.

Next, you place a magic comment directly before each package-level variable that holds the contents of a file. This comment must must start with `go:embed`, with no space between the slashes and `go:embed`. The comment must also be on the line directly before the variable. (technically, it is legal to have blank lines or other, non-magic comments between the embedding comment and the variable declaration, but don't do it) In our sample, we are embedding the contents of `passwords.txt` into the package-level variable named `passwords`. It is idiomatic to treat a variable with an embedded value as immutable. As mentioned earlier, you can only embed into a package-level variable. The variable must be of type `string`, `[]byte`, or `embed.FS`. If you have a single file, it's simplest to use `string` or `[]byte`.

If you need to place one or more directories of files into your program, use a variable of type `embed.FS`. This type implements three interfaces defined in the `io/fs` package, `FS`, `ReadDirFS`, and `ReadFileFS` . This allows an instance of `embed.FS` to represent a virtual file system. There is a sample project with `embed.FS` in the `sample_code/help_system` directory in the chapter 11 repository. This program provides a simple command line help system. If you don't provide a help file, it lists all available files. If you specify a file that's not present, it returns an error.

```
package main

import (
    "embed"
    "fmt"
    "io/fs"
    "os"
    "strings"
)

//go:embed help
var helpInfo embed.FS

func main() {
    if len(os.Args) == 1 {
        printHelpFiles()
```

```
            os.Exit(0)
        }
        data, err := helpInfo.ReadFile("help/" + os.Args[1])
        if err != nil {
            fmt.Println(err)
            os.Exit(1)
        }
        fmt.Println(string(data))
    }
```

Here's the output when you build and run this program:

```
$ go build
$ ./help_system
contents:
advanced/topic1.txt
advanced/topic2.txt
info.txt

$ ./help_system advanced/topic1.txt
This is advanced topic 1.

$ ./help_system advanced/topic3.txt
open help/advanced/topic3.txt: file does not exist
```

There are a couple of things to notice. First, we no longer need to use a blank import for `embed`, since we are using `embed.FS`. Second, the directory name is part of the file system that's embedded. The users of this program don't enter the "help/" prefix, so we have to prepend it in the call to `ReadFile`.

The `printHelpFiles` function shows how we can treat an embedded virtual file system just like a real one:

```
    func printHelpFiles() {
        fmt.Println("contents:")
        fs.WalkDir(helpInfo, "help",
            func(path string, d fs.DirEntry, err error) error {
                if !d.IsDir() {
                    _, fileName, _ := strings.Cut(path, "/")
                    fmt.Println(fileName)
                }
```

```
            return nil
        })
}
```

We use the `WalkDir` function in `io/fs` to walk through embedded file system. `WalkDir` takes in an instance of `fs.FS`, a path to start at, and a function. This function is called for every file and directory in the file system, starting from the specified path. If the `fs.DirEntry` is not a directory, we print out its full path name, removing the `help/` prefix using `strings.Cut`.

There are a few more things to know about file embedding. While all of our examples have been text files, you can embed binary files as well. You can also embed multiple files or directories into a single `embed.FS` variable by specifying their names, separated by spaces. When embedding a file or directory that has a space in its name, put the name in quotes.

In addition to exact file and directory names, you can use wildcards and ranges to specify the names of the files and directories you want to embed. The syntax is defined at *https://pkg.go.dev/path@go1.19.5#Match* , but it follows common conventions. For example, `*` matches 0 or more characters and `?` matches a single character.

All embedding specifications, whether or not they use match patterns, are checked by the compiler. If they aren't valid, compilation fails. Here are the ways a pattern can be invalid:

- If the specified name or pattern doesn't match a file or directory
- If you specify multiple file names or patterns for a `string` or `[]byte` variable
- If you specify a pattern for a `string` or `[]byte` variable and it matches more than one file

Embedding hidden files

Including files in a directory tree that start with `.` or _ is a little complicated. Many operating systems consider these to be hidden files, so they are not included by default when a directory name is specified. However, there are two ways to override this behavior. The first is to put `/*` after the name of a directory you want to embed. This will include all hidden files within the root directory, but it will not include hidden files in its subdirectories. To include all hidden files in all subdirectories, put `all:` before the name of the directory.

This sample program (which can be found in the `sample_files/embed_hidden` directory in the chapter 11 repository) makes this easier to understand. In our sample the directory `parent_dir` contains two files `.hidden` and `visible`, and one subdirectory, `child_dir`. The `child_dir` subdirectory contains two files, `.hidden` and `visible`.

Here is the code for our program:

```
//go:embed parent_dir
var noHidden embed.FS

//go:embed parent_dir/*
var parentHiddenOnly embed.FS

//go:embed all:parent_dir
var allHidden embed.FS

func main() {
    checkForHidden("noHidden", noHidden)
    checkForHidden("parentHiddenOnly", parentHiddenOnly)
    checkForHidden("allHidden", allHidden)
}

func checkForHidden(name string, dir embed.FS) {
    fmt.Println(name)
    allFileNames := []string{
        "parent_dir/.hidden",
        "parent_dir/child_dir/.hidden",
    }
    for _, v := range allFileNames {
        _, err := dir.Open(v)
        if err == nil {
            fmt.Println(v, "found")
        }
```

```
        }
        fmt.Println()
    }
```

The output of the program is:

```
noHidden

parentHiddenOnly
parent_dir/.hidden found

allHidden
parent_dir/.hidden found
parent_dir/child_dir/.hidden found
```

go generate

The `go generate` tool is a little different, because it doesn't do anything by itself. When you run `go generate`, it looks for specially formatted comments in your source code and runs programs specified in those comments. While you could use `go generate` to run anything at all, it is most commonly used by developers to run tools that (unsurpringly, given the name) generate source code. This could be from analyzing existing code and adding functionality or processing schemas and making source code out of it.

A good example of something that can be automatically converted to code are Protocol Buffers, sometimes called *protobufs*. Protobuf is a popular binary format that is used by Google to store and transmit data. When working with protobufs, you write a *schema*, which is a language-independent description of the data structure. Developers who want write programs to interact with data in protobuf format run tools that process the schema and produce language-specific data structures to hold the data and language-specific functions to read and write data in protobuf format.

Let's see how this works in Go. You can find a sample module in the proto_generate repo. The module contains a protobuf schema file called `person.proto`:

```
syntax = "proto3";

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
```

While making a struct that implements `Person` would be easy, writing the code to convert back and forth from the binary format is difficult. Let's use tools from Google to do the hard work and invoke them with `go generate`. You need to install two things. The first is the `protoc` binary for your computer. You can find installation instructions at *https://grpc.io/docs/protoc-installation/* . Next, use `go install` to install the Go protobuf plugins:

```
$ go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28
```

In `main.go`, there is the magic comment that's processed by `go generate`:

```
//go:generate protoc -I=. --go_out=.
  --go_opt=module=github.com/learning-go-book-2e/proto_generate
  --go_opt=Mperson.proto=github.com/learning-go-book-2e/proto_generate/data
  person.proto
```

Run `go generate` by typing:

```
$ go generate ./...
```

After running `go generate`, you'll see a new directory called `data` that contains a file named `person.pb.go`. It contains the source code for the `Person` struct, and some methods and functions that are used by the `Marshal` and `Unmarshal` functions in the `google.golang.org/protobuf/proto` module. We call these functions in our `main` function:

```
func main() {
    p := &data.Person{
        Name: "Bob Bobson",
        Id: 20,
        Email: "bob@bobson.com",
    }
    fmt.Println(p)
    protoBytes, _ := proto.Marshal(p)
    fmt.Println(protoBytes)
    var p2 data.Person
    proto.Unmarshal(protoBytes,&p2)
    fmt.Println(&p2)
}
```

Build and run the program as usual:

```
$ go build
$ ./proto_generate
name:"Bob Bobson"  id:20  email:"bob@bobson.com"
[10 10 66 111 98 32 66 111 98 115 111 110 16 20 26 14 98
  111 98 64 98 111 98 115 111 110 46 99 111 109]
name:"Bob Bobson"  id:20  email:"bob@bobson.com"
```

Another tool commonly used with `go generate` is `stringer`. As we discussed in Chapter 7, enumerations in Go lack many of the features that are found in other languages with enumerations. One of those features is automatically generating a printable name for each value in the enumeration. The `stringer` tool is used with `go generate` to add a `String` method to your enumeration's values so they can be printed.

Install `stringer` with `go install golang.org/x/tools/cmd/stringer@latest` . The `sample_code/stringer_demo` directory in the chapter 11 repository provides a very simple example of how to use `stringer`. Here's the source in `main.go`:

```
type Direction int

const (
    _ Direction = iota
    North
    South
```

```
    East
    West
)

//go:generate stringer -type=Direction

func main() {
    fmt.Println(North.String())
}
```

Run `go generate ./...` and you'll see a new file generated called `direction_string.go`. Use `go build` to build the `string_demo` binary and when you run it, you'll get the output:

```
North
```

You can configure `stringer` and its output in multiple ways. Arjun Mahishi has written a great blog post that describes how to use `stringer` and customize its output.

go generate and Makefiles

Since the job of `go generate` is to run other tools, you might wonder if it's worth using when you have a perfectly good Makefile in your project. The advantage of `go generate` is that it creates a separation of responsibilities. Use `go generate` commands to mechanically create source code, and use the Makefile to validate and compile source code.

It is a best practice to commit the source code created by `go generate` to version control. (The sample projects don't include generated source code so you could see it work.) This allows people browsing your source code to see everything that's invoked, even the generated parts.

Checking in your generated source code technically means that you don't *need* to run `go generate` unless it will produce different output, such as processing a modified protobuf definition or an updated enumeration. However, it's still a good idea to automate calling `go generate` before `go build`. Relying on a manual process is asking for trouble. Some generator

tools, like `stringer`, include clever tricks to block compilation if you forget to re-run `go generate`, but that's not universal. You'll inevitably waste time during testing trying to understand why a change didn't show up before realizing that you forgot to invoke `go generate`. (I made this mistake multiple times before I learned my lesson.) Given this, it is best to add a `generate` step to your `Makefile` and make it a dependency of your `build` step.

There are two situations where I would disregard this advice. The first is if invoking `go generate` on identical input produces source files with minor differences (such as a timestamp). A well-written `go generate` tool should produce identical output every time it's run on the same input, but there are no guarantees that every tool you need to use is well-written. You don't want to keep on checking in new versions of files that are functionally identical, as they will clutter your version control system and make your code reviews more noisy.

The second situation is if it `go generate` a very long time to complete. Fast builds are a feature of Go, because they allow developers to stay focused and get rapid feedback. If you are noticeably slowing down a build to generate identical files, the loss in developer productivity is not worth it. In both cases, all you can do is leave lots of comments to remind people to rebuild when things change and hope that everyone on your team is diligent.

Reading the build info inside a Go binary

As companies develop more of their own software, it is becoming increasingly common for them to want to understand exactly what they have deployed to their data centers and cloud environments, down to the version and the dependencies. You might wonder why you'd want to get this information from compiled code. After all, a company already has this information in version control. For companies with mature development and deployment pipelines, they can capture this information right before deploying a program, allowing them to be sure that the information is

accurate. However, many, if not most, companies don't track exactly what version of internal software was deployed. In some cases, software can be deployed for years without being replaced and no one remembers much about it. If a vulnerability is reported in a version of a third party library, you need to either find some way to scan your deployed software and figure out what versions of third party libraries are deployed, or redeploy everything just to be safe. In the Java world, this exact problem happened when a serious vulnerability was discovered in the very popular Log4J library.

Luckily, Go solves this problem for you. Every Go binary you create with `go build` automatically contains build information on not only what versions of what modules make up the binary, but what build commands were used, what version control system was used, and what revision the code was at in your version control system. You can view this information with the `go version -m` command. Let's show what its output is for the `vulnerable` program when built on an Apple Silicon Mac:

```
$ go build
go: downloading gopkg.in/yaml.v2 v2.2.7
$ go version -m vulnerable
vulnerable: go1.19
    path     github.com/learning-go-book-2e/vulnerable
    mod      github.com/learning-go-book-2e/vulnerable    (devel)
    dep      gopkg.in/yaml.v2  v2.2.7
h1:VUgggvou5XRW9mHwD/yXxIYSMtY0zoKQf/v...
    build    -compiler=gc
    build    CGO_ENABLED=1
    build    CGO_CFLAGS=
    build    CGO_CPPFLAGS=
    build    CGO_CXXFLAGS=
    build    CGO_LDFLAGS=
    build    GOARCH=arm64
    build    GOOS=darwin
    build    vcs=git
    build    vcs.revision=623a65b94fd02ea6f18df53afaaea3510cd1e611
    build    vcs.time=2022-10-02T03:31:05Z
    build    vcs.modified=false
```

Because this information is embedded into every binary, `govulncheck` is capable of scanning go programs to check for libraries with known vulnerabilities:

```
$ govulncheck vulnerable
govulncheck is an experimental tool. Share feedback at
https://go.dev/s/govulncheck-feedback.

Scanning for dependencies with known vulnerabilities...
Found 1 known vulnerability.

Vulnerability #1: GO-2020-0036
  Due to unbounded aliasing, a crafted YAML file can cause
  consumption of significant system resources. If parsing user
  supplied input, this may be used as a denial of service vector.
  Found in: gopkg.in/yaml.v2@v2.2.7
  Fixed in: gopkg.in/yaml.v2@v2.2.8
  More info: https://pkg.go.dev/vuln/GO-2020-0036
```

Be aware that `govulncheck` can't track down exact lines of code when inspecting a binary. If `govulncheck` finds a problem in a binary, use `go version -m` to find out the exact deployed version, check the code out of version control, and run it again against the source code to pinpoint the issue.

If you want to build your own tools to read the build information, look at the debug/buildinfo package in the standard library.

Building Go Binaries for other platforms

One of the advantages of a VM-based language like Java, JavaScript, or Python is that you can take your code and get it to run on any computer where the virtual machine has been installed. This portability makes it easy for developers using these languages to build programs on a Windows or Mac laptop and deploy it on a Linux server, even though the operating system and possibly the CPU architecture are different.

Go programs are compiled to native code, which means that the generated binary is only compatible with a single operating system and CPU architecture. However, that doesn't mean that Go developers need to maintain a menagerie of machines (virtual or otherwise) to release on multiple platforms. The `go build` command makes it very easy to *cross-compile*, or create a binary for a different operating system and/or CPU. When `go build` is run, the target operating system is specified by the `GOOS` environment variable. Similarly, the `GOARCH` environment variable specifies the CPU architecture. If you don't set them explicitly, `go build` defaults to using the values for your current computer, which is why you've never had to worry about these variables before.

The valid values and combinations for `GOOS` and `GOARCH` (sometimes pronounced "GOOSE" and "GORCH") are found on *https://go.dev/doc/install/source#environment* . Some of the supported operating systems and CPUs are a bit esoteric and others might require some translation. For example, `darwin` refers to MacOS (Darwin is the name of kernel of Mac OS), and `amd64` mean 64-bit Intel compatible CPUs.

Let's go back to our `vulnerable` program one last time. When using an Apple Silicon Mac (which has an ARM64 CPU), running `go build` defaults to `darwin` for `GOOS` and `arm64` for `GOARCH`. You can confirm this using the `file` command:

```
$ go build
$ file vulnerable
vulnerable: Mach-O 64-bit executable arm64
```

Here is how to build a binary for Linux on 64-bit Intel CPUs:

```
$ GOOS=linux GOARCH=amd64 go build
$ file vulnerable
vulnerable: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, Go BuildID=IDHVCE8XQPpWluGpMXpX/4VU3GpRZEifN8TzUrT_6/1c30VcDYNVPfSSN-
zCkz/JsZSLAbWkxqIVhPkC5p5, with debug_info, not stripped
```

Build Tags

When writing programs that need to run on multiple operating systems or CPU architectures, you sometimes need different code for different platforms. You also might want to write a module that takes advantage of the latest Go features, but is still backwards compatible with older Go compilers.

There are two ways to create targeted code. The first is to use the name of the file to indicate when the file should be included in the build. You do this by adding the target GOOS and GOARCH, separated by _, to the file name before `.go`. For example, if you have a file that you only want to be compiled on Windows, you'd name the file `something_windows.go`, but if you only wanted it to be compiled when building for ARM64 Windows, name the file `something_windows_arm64.go`

A *build tag* (also called a *build constraint*) is the other option you can use to specify when a file is compiled. Like embedding and generating, build tags take advantage of a magic comment. In this case, it's `//go:build` . This comment must be placed on the line before the package declaration in your file.

Build tags use boolean operators (||, &&, and !) and parenthesis to specify exact build rules for architectures, operating systems, and go versions. The build tag `//go:build (!darwin && !linux) || (darwin && !go1.12)` (which really appears in the Go standard library) specifies that the file should not be compiled on Linux or Mac OS, except it's OK to compile it on Mac OS if the Go verion is 1.11 or earlier.

There are also some meta build constraints available. The constraint `unix` matches any Unix-ish platform, and `cgo` matches if cgo is supported by the current platform and is enabled.(We cover cgo in Chapter 16.)

The question becomes when you should use file names to indicate where to run code and when you should use build tags. Because build tags allow binary operators, you can specify a more specific set of platforms with them. The Go standard library sometimes takes a belt-and-suspenders

approach. The package *internal/cpu* in the standard library has platform-specific source code for CPU feature detection. The file *internal/cpu/cpu_arm64_darwin.go* has a name that indicates that it is only meant for computers using Apple CPUs. It also has a `//go:build arm64 && darwin && !ios` line in the file to indicate that it should only be compiled when building for Apple Silicon Macs and not for iPhones or iPads. The build tags are able to specify the target platform with more detail, but following the file name convention makes it easy for a person to find the right file for a given platform.

In addition to the built-in build tags that represent go versions, operating systems and CPU architectures, you can also use any string at all as a custom build tag. You can then control compilation of that file with the `-tags` command line flag. For example, if you put `//go:build gopher` on the line before the package declaration in a file, it will not be compiled unless you include a `-tags gopher` flag as part of the `go build`, `go run`, or `go test` command.

Custom build tags are surprisingly handy. If you have a source file that you don't want to build right now (perhaps it doesn't compile yet, or it's an experiment that's not ready to be included), it is idiomatic to skip over the file by putting `//go:build ignore` on the line before the package declaration. We will see another use for custom build tags when looking at integration tests in Chapter 15.

Testing versions of Go

Despite Go's strong backward compatibility guarantees, bugs do happen. It's natural to want to make sure that a new release doesn't break your programs. You also might get a bug report from a user of your library saying that your code doesn't work as expected on an older version of Go. One option is to install a secondary Go environment. For example, if you wanted to try out version 1.19.2, you would use the following commands:

```
$ go install golang.org/dl/go1.19.2@latest
$ go1.19.2 download
```

You can then use the command `go1.19.2` instead of the `go` command to see if version 1.19.2 works for your programs:

```
$ go1.19.2 build
```

Once you have validated that your code works, you can uninstall the secondary environment. Go stores secondary Go environments in the *sdk* directory within your home directory. To uninstall, delete the environment from the *sdk* directory and the binary from *go/bin* directory. Here's how to do that on Mac OS, Linux, and BSD:

```
$ rm -rf ~/sdk/go.19.2
$ rm ~/go/bin/go1.19.2
```

Use go help to learn more about Go tooling

You can learn more about Go's tooling and runtime environment with the `go help` command. It contains exhaustive information about all of the commands mentioned here, as well as things like modules, import path syntax, and working with non-public source code. For example, for details on the environment variables listed by `go env`, type `go help environment`.

Exercises

These exercises cover some of the tools that you've learned about in this chapter. You can find the solutions in the `exercise_solutions` directory in the chapter 11 repository.

1. Go to *https://www.ohchr.org/en/human-rights/universal-declaration/translations/english* and copy the text of the Universal Declaration of Human Rights into a text file called english_rights.txt. Click the "Other Languages" link and copy the text of the document in a few additional languages into files named LANGUAGE_rights.txt. Create a program that embeds these files into package-level variables. Your program should take in one command-line parameter, the name of a language. It should then print out the UDHR in that language.
2. Use `go install` to install `staticcheck`. Run it against your program and fix any problems it finds.
3. Cross-compile your program for ARM64 on Windows. If you are using an ARM64 Windows computer, cross-compile for AMD64 on Linux.

Wrapping Up

In this chapter, we learned about the tools that Go provides to improve software engineering practice. We also looked at third-party code-quality tools, and briefly covered containerized software. In the next chapter, we're going to explore one of the signature features in Go: concurrency.

About the Author

Jon Bodner has been a software engineer, lead developer, and architect for over 25 years. In that time, he has worked on software across many fields, including education, finance, commerce, healthcare, law, government, and internet infrastructure.

Jon is currently a Staff Engineer at Datadog, working on improving the onboarding experience for customers.

Previously, Jon was a Senior Distinguished Engineer at Capital One, where he worked on commercialization of technology, contributed to the company's development and testing workflow, developed patented techniques for web payment page detection and population, and coauthored tools for finding and managing software development issues.

Jon is a frequent speaker at Go conferences, and his blog posts on Go and software engineering have been viewed more than 300,000 times. He is the creator of the Proteus data access library and codeveloper of checks-out, a fork of the LGTM project.