CHAPTER 1

# Setting Up Your Go Environment

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, you 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. You can find the latest version of the tools 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.

> 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 go1.20.5.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 go1.20.5.linux-amd64.tar.gz`.

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 Geert Baeke's blog post "Distroless or Scratch for Go Apps".

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 go1.20.5 darwin/arm64
```

This tells you that this is Go version 1.20.5 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 go1.20.5 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 Chapters 10, 11, and

15. For now, let's take a quick look at the most commonly used tools by writing everyone's favorite first application: "Hello, World!""

Since the introduction of Go in 2009, several changes have occurred in the way 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, you 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 you need to do is create a directory to hold your program. Call it *ch1*. On the command line, enter the new directory. If your computer's terminal uses bash or zsh, this looks like the following:

```
$ 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 10, 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 you. The contents of a basic *go.mod* file look like this:

```
module hello_world

go 1.20
```

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 10.

## go build

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

```go
package main

import "fmt"

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

(Yes, the indentation in this code example looks sloppy: I meant to do that! You will see why in just a bit.)

Let's quickly go over the parts of the Go file you 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. You're using a function in the `fmt` (usually pronounced "fumpt") package from the standard library, so you list the package here. Unlike other languages, Go imports only 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. You 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 you 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 you wanted to compile the code to a binary called "hello," you would use the following:

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

In "Using go run to Try Out Small Programs" on page 263, I 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 in the way 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 versus 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.

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 the following:

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

Using `./...` tells a Go tool to apply the command to all the files in the current directory and all subdirectories. You will see it again as you 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.

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 does *only* **`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 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:

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

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

```go
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

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

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

`fmt.Printf` is 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, you 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` detects is whether a value exists for every placeholder in a formatting template. Run `go vet` on the 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 the bug, you can easily fix it. Change line 6 in *hello.go* to:

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

While `go vet` catches several common programming errors, there are things that it cannot detect. Luckily, third-party Go code-quality tools can close the gap. Some of the most popular code-quality tools are covered in "Using Code-Quality Scanners" on page 267.

Just as you should run `go fmt` to make sure your code is formatted properly, run `go vet` to scan for possible bugs in valid code. These commands 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 you 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. Excellent Go development tools are available 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.

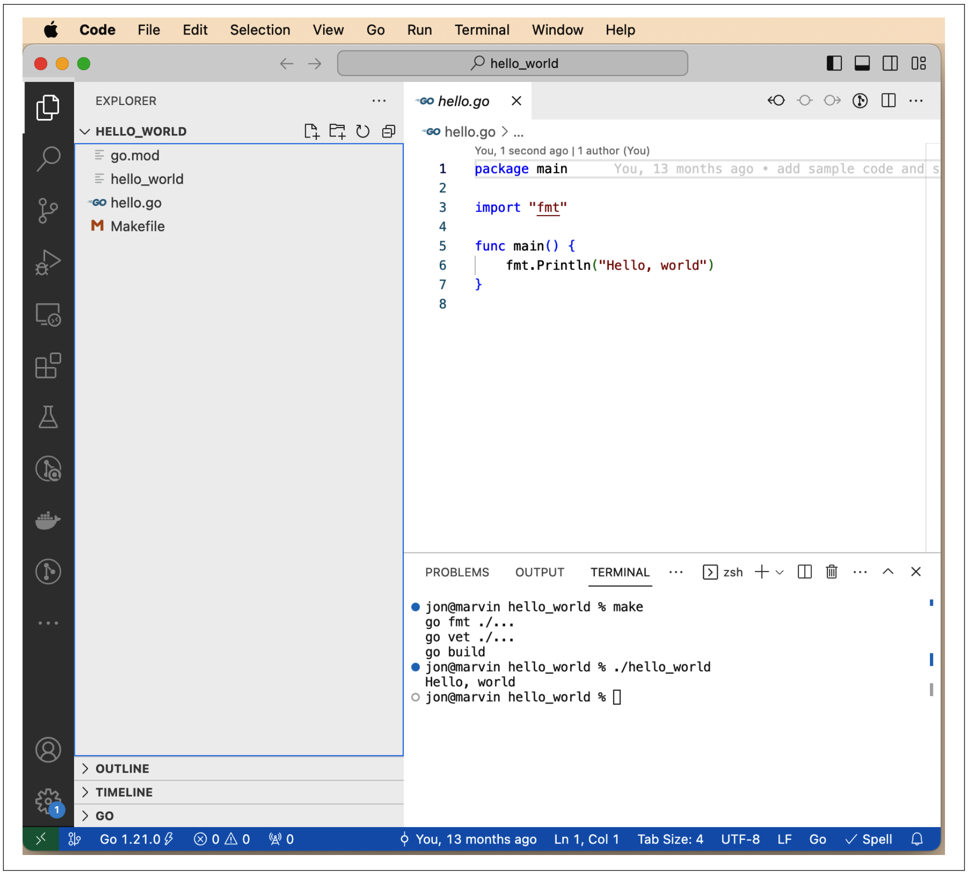
### Visual Studio Code

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.

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.



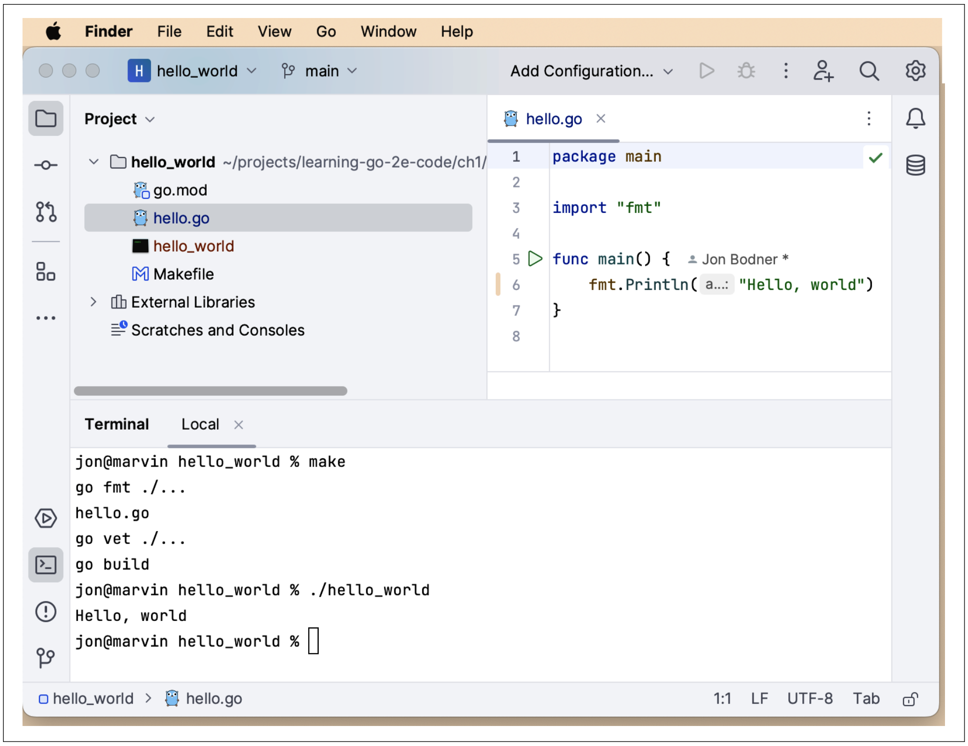
```go
package main

import "fmt"

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

```
jon@marvin hello_world % make
go fmt ./...
go vet ./...
go build
jon@marvin hello_world % ./hello_world
Hello, world
jon@marvin hello_world % 
```

*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 plug-in to get it to work.



```go
package main

import "fmt"

func main() {   Jon Bodner *
	fmt.Println( a...: "Hello, world")
}
```

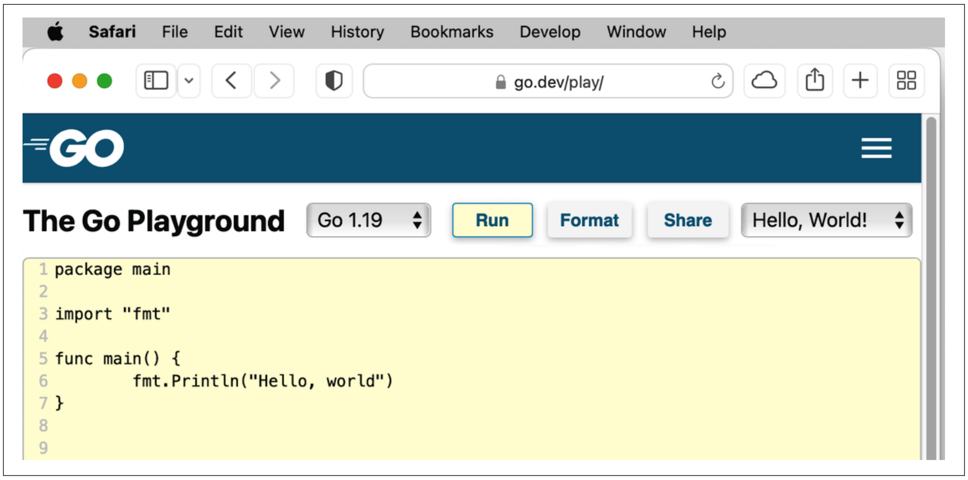
```
jon@marvin hello_world % make
go fmt ./...
hello.go
go vet ./...
go build
jon@marvin hello_world % ./hello_world
Hello, world
jon@marvin hello_world %
```

*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, a 30-day free trial is 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 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).



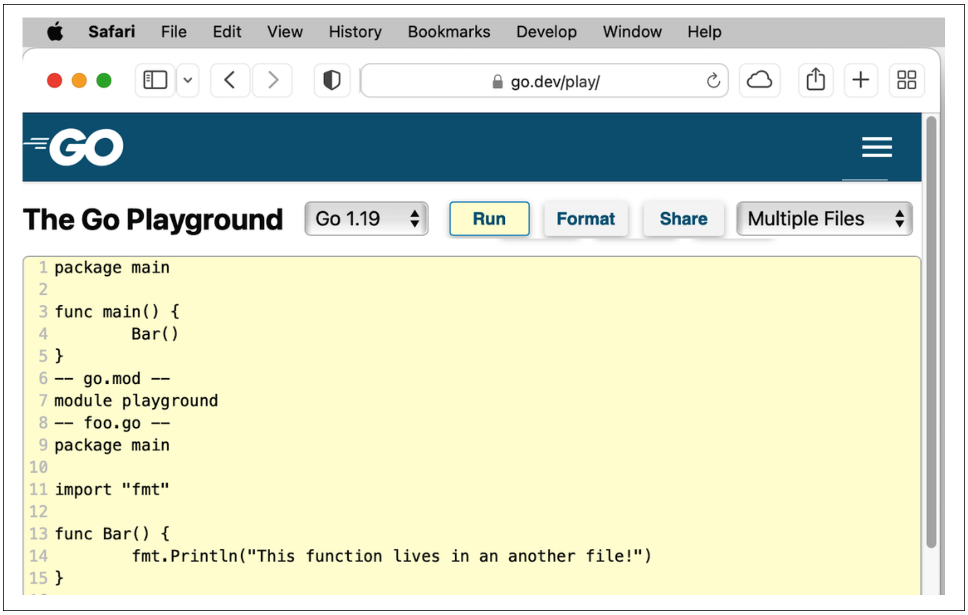
```go
package main

import "fmt"

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


```

*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 make network connections only 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 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.

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.



```
package main

func main() {
	Bar()
}
-- go.mod --
module playground
-- foo.go --
package main

import "fmt"

func Bar() {
	fmt.Println("This function lives in an another file!")
}
```

*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, figuring out what's going on is not too difficult. Each possible operation is called a *target*. The `.DEFAULT_GOAL` defines which target is run when no target is specified. In this case, the default is the `build` target. Next you 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 a directory or file in your project has the same name as one of the listed targets.

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

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

Entering a single command formats the code correctly, checks it for nonobvious errors, and compiles it. You 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 Chapter 1 repository for this book.

## The Go Compatibility Promise

As with all programming languages, the Go development tools are periodically updated. Since Go 1.2, a new release has occurred roughly every six months. Patch releases with bug and security fixes are also 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 go1.20.6.linux-amd64.tar.gz
$ rm -rf /usr/local/old-go
```

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

Each chapter has exercises at the end to let you try out the ideas that I cover. You can find answers to these exercises in the Chapter 1 repository.

1. Take the "Hello, world!" program and run it on The Go Playground. Share a link to the code in the playground with a coworker 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, you learned how to install and configure your Go development environment. You also learned about tools for building Go programs and ensuring code quality. Now that your environment is ready, you're on to the next chapter, where you'll explore the built-in types in Go and how to declare variables.