O'REILLY®

Learning Go

An Idiomatic Approach to Real-World Go Programming

Jon Bodner

O'REILLY®

Learning Go

Go is rapidly becoming the preferred language for building web services. While there are plenty of tutorials available that teach Go's syntax to developers with experience in other programming languages, tutorials aren't enough. They don't teach Go's idioms, so developers end up recreating patterns that don't make sense in a Go context. This practical guide provides the essential background you need to write clear and idiomatic Go.

No matter your level of experience, you'll learn how to think like a Go developer. Author Jon Bodner introduces the design patterns experienced Go developers have adopted and explores the rationale for using them. You'll also get a preview of Go's upcoming generics support and how it fits into the language.

- Learn how to write idiomatic code in Go and design a Go project
- Understand the reasons for the design decisions in Go
- Set up a Go development environment for a solo developer or team
- Learn how and when to use reflection, unsafe, and cgo
- Discover how Go's features allow the language to run efficiently
- Know which Go features you should use sparingly or not at all

Jon Bodner is a software engineer, lead developer, and architect with over 20 years of experience. In that time, he's worked on software across fields including education, finance, commerce, healthcare, law, government, and internet infrastructure. Jon is a distinguished engineer at Capital One.

"Jon has written the programmers' guide to learning Go. It strikes just the right balance of giving a good overview of what you need to know without rehashing well understood concepts from other languages."

—**Steve Francia**
Go language product lead at Google, and creator of Hugo, Cobra & Viper

"Go is unique and even experienced programmers have to unlearn a few things and think differently about software. *Learning Go* does a good job of working through the big features of the language while pointing out idiomatic code, pitfalls, and design patterns along the way."

—**Aaron Schlesinger**
Sr. Engineer, Microsoft

OTHER PROGRAMMING LANGUAGES

US $59.99 CAN $79.99
ISBN: 978-1-492-07721-3

Twitter: @oreillymedia
facebook.com/oreilly

Praise for *Learning Go*

"Go is unique and even experienced programmers have to unlearn a few things and think differently about software. *Learning Go* does a good job of working through the big features of the language while pointing out idiomatic code, pitfalls, and design patterns along the way."

—Aaron Schlesinger, Sr. Engineer, Microsoft

"Jon has been a critical voice in the Go community for many years and we have been strongly benefitted from his talks and articles. With *Learning Go*, Jon has written the programmers' guide to learning Go. It strikes just the right balance of giving a good overview of what you need to know without rehashing well understood concepts from other languages."

—Steve Francia, Go language product lead, Google, and author of Hugo, Cobra, and Viper

"Bodner gets Go. In clear, lively prose, he teaches the language from its basics to advanced topics like reflection and C interop. He demonstrates through numerous examples how to write *idiomatic* Go, with its emphasis on clarity and simplicity. He also takes the time to explain the underlying concepts that can affect your program's behavior, like the effects of pointers on memory layout and garbage collection. Beginners who read this book will come up to speed quickly, and even experienced Go programmers are likely to learn something."

—Jonathan Amsterdam, Software Engineer on the Go team at Google

"*Learning Go* is the essential introduction to what makes the Go programming language unique as well as the design patterns and idioms that make it so powerful. Jon Bodner manages to connect the fundamentals of the language to Go's philosophy, guiding readers to write Go the way it was meant to be written."

—Robert Liebowitz, Software Engineer at Morning Consult

"Jon wrote a book that does more than just reference Go; it provides an idiomatic and practical understanding of the language. Jon's industry experience is what drives this book, and it will help those looking to be immediately productive in the language."

—William Kennedy, Managing Partner at Ardan Labs

FIRST EDITION

Learning Go

An Idiomatic Approach to Real-World Go Programming

Jon Bodner

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

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: Michele Cronin
Production Editor: Beth Kelly
Copyeditor: Piper Editorial Consulting, LLC
Proofreader: Piper Editorial Consulting, LLC

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

March 2021: First Edition

Revision History for the First Edition
2021-03-02: First Release
2021-05-14: Second Release

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

978-1-492-07721-3

[LSI]

Table of Contents

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 generics (yet), no aspect-oriented programming, no function overloading, and certainly no operator overloading. There's no pattern matching, no named parameters, no exceptions. To the horror of many, there are *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.

This element signifies a tip or suggestion.

This element signifies a general note.

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

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

Every programming language needs a development environment, and Go is no exception. If you've already written a Go program or two, you should 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; installing Go and its supporting tools is easy. After we set up our environment and verify it, we'll 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 write Go code, you first 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.

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 FreeBSD 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 go1.15.2.linux-amd64.tar.gz
$ echo 'export PATH=$PATH:/usr/local/go/bin' >> $HOME/.profile
$ source $HOME/.profile
```

Go programs compile to a single binary and do not require any additional software to be installed in order to run them. Install the Go development tools only on computers that build Go programs.

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 go1.15.2 darwin/amd64
```

This tells you that this is Go version 1.15.2 on Mac OS. (Darwin is the name of the kernel for Mac OS and amd64 is the name for the 64-bit CPU architecture from both AMD and Intel.)

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 Mac OS and other Unix-like systems, use `which go` to see the `go` command being executed, if any. If it isn't the `go` command at */usr/local/go/bin/go*, you need to fix your executable path.

If you're on Linux or FreeBSD, 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.

The Go Workspace

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 modern Go development, the rule is simple: you are free to organize your projects as you see fit.

However, Go still expects there to be a single workspace for third-party Go tools installed via `go install` (see "Getting Third-Party Go Tools" on page 5). By default, this workspace is located in *$HOME/go*, with source code for these tools stored in *$HOME/go/src* and the compiled binaries in *$HOME/go/bin*. You can use this default or specify a different workspace by setting the `$GOPATH` environment variable.

Whether or not you use the default location, it's a good idea to explicitly define `GOPATH` and to put the *$GOPATH/bin* directory in your executable path. Explicitly defining `GOPATH` makes it clear where your Go workspace is located and adding *$GOPATH/bin* to your executable path makes it easier to run third-party tools installed via `go install`, which we'll talk about in a bit.

If you are on a Unix-like system using `bash`, add the following lines to your *.profile*. (If you are using `zsh`, add these lines to *.zshrc* instead):

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

You'll need to `source` *$HOME/.profile* to make these changes take effect in your current terminal window.

On Windows, run the following commands at the command prompt:

```
setx GOPATH %USERPROFILE%\go
setx path "%path%;%USERPROFILE%\bin"
```

After running these commands, you must close your current command prompt and open a new one for these changes to take effect.

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 env` command. Many of them control low-level behavior that can be safely ignored, but we cover some of these variables when discussing modules and cross-compilation.

Some online resources tell you to set the GOROOT environment variable. This variable specifies the location where your Go development environment is installed. This is no longer necessary; the `go` tool figures this out automatically.

The go Command

Out of the box, Go ships with many development tools. You access these tools via the `go` command. They include a compiler, code formatter, linter, dependency manager, test runner, and more. As we learn how to build high-quality idiomatic Go, we'll explore many of these tools throughout the book. Let's start with the ones that we use to build Go code and use the `go` command to build a simple application.

go run and go build

There are two similar commands available via `go`: `go run` and `go build`. Each takes either a single Go file, a list of Go files, or the name of a package. We are going to create a simple program and see what happens when we use these commands.

go run

We'll start with `go run`. Create a directory called *ch1*, open up a text editor, enter the following text, and save it inside *ch1* to a file named *hello.go*:

```
package main

import "fmt"

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

After the file is saved, open up a terminal or command prompt and type:

```
go run hello.go
```

You should see `Hello, world!` printed in the console. 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. You might be thinking: I thought Go was a compiled language. What's going on?

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.

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

go build

Most of the time you want to build a binary for later use. That's where you use the `go build` command. On the next line in your terminal, type:

```
go build hello.go
```

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

The name of the binary matches the name of the file or package that you passed in. 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_world," we would use:

```
go build -o hello_world hello.go
```

Use `go build` to create a binary that is distributed for other people to use. Most of the time, this is what you want to do. Use the `-o` flag to give the binary a different name or location.

Getting Third-Party Go Tools

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 into your Go workspace via the `go install` command.

Go's method for publishing code is a bit different than most other languages. Go developers don't rely on a centrally hosted service, like Maven Central for Java or the NPM registry for JavaScript. Instead, they share projects via their source code repositories. The `go install` command takes an argument, which is the location of the source code repository of the project you want to install, 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 into your *$GOPATH/bin* directory.

Let's look at a quick example. There's 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 in your *$GOPATH/bin* directory.

As we'll talk about in "Module Proxy Servers" on page 200, 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://www.golang.org

Summary:
  Total:        0.6864 secs
  Slowest:      0.3148 secs
```

```
Fastest:      0.0696 secs
Average:      0.1198 secs
Requests/sec: 291.3862
```

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 tools written in Go in your Go workspace; 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 a very convenient way to distribute Go programs to other Go developers.

Formatting Your Code

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 laid out. 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 One True 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.

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 has publicly stated that better tooling was his original motivation.

The Go development tools include a command, `go fmt`, which automatically reformats your code to match the standard format. It does things like fixing up the whitespace for indentation, lining up the fields in a struct, and making sure there is proper spacing around operators.

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.

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 never put the semicolons in themselves; the Go compiler does it for them 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 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.

Always run `go fmt` or `goimports` before compiling your code!

Linting and Vetting

While `go fmt` ensures your code is formatted correctly, it's just the first step in ensuring that your code is idiomatic and of high quality. 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.

There are tools that help to enforce this style. The first is called `golint`. (The term "linter" comes from the Unix team at Bell Labs; the first linter was written in 1978.) It tries to ensure your code follows style guidelines. Some of the changes it suggests include properly naming variables, formatting error messages, and placing comments on public methods and types. These aren't errors; they don't keep your programs from compiling or make your program run incorrectly. Also, you cannot automatically assume that `golint` is 100% accurate: because the kinds of issues that `golint` finds are more fuzzy, it sometimes has false positives and false negatives. This means that you don't *have* to make the changes that `golint` suggests. But you should take the suggestions from `golint` seriously. Go developers expect code to look a certain way and follow certain rules, and if your code does not, it sticks out.

Install `golint` with the following command:

```
go install golang.org/x/lint/golint@latest
```

And run it with:

```
golint ./...
```

That runs `golint` over your entire project.

There is another class of errors that developers run into. The code is syntactically valid, but there are mistakes that are not what you meant to do. This includes things like passing the wrong number of parameters to formatting methods or assigning values to variables that are never used. The `go` tool includes a command called `go vet` to detect these kinds of errors. Run `go vet` on your code with the command:

```
go vet ./...
```

There are additional third-party tools to check code style and scan for potential bugs. However, running multiple tools over your code slows down the build because each tool spends time scanning the source code for itself. Rather than use separate tools, you can run multiple tools together with `golangci-lint`. It combines `golint`, `go`

`vet`, and an ever-increasing set of other code quality tools. Once it is installed, you run `golangci-lint` with the command:

```
golangci-lint run
```

Because `golangci-lint` runs so many tools (as of this writing, it runs 10 different linters by default and allows you to enable another 50), it's inevitable that your team may disagree with some of its suggestions. You can configure which linters are enabled and which files they analyze by including a file named *.golangci.yml* at the root of your project. Check out the documentation for the file format.

I recommend that you start off using `go vet` as a required part of your automated build process and `golint` as part of your code review process (since `golint` might have false positives and false negatives, 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.

Make `golint` and `go vet` (or `golangci-lint`) part of your development process to avoid common bugs and nonidiomatic code. But if you are using `golangci-lint`, make sure your team agrees on the rules that you want to enforce!

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. Luckily, 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 tools. 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 development kit 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, linting, and finding usages. You can check out the language server protocol.

Once your tools are set up, you can then 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.

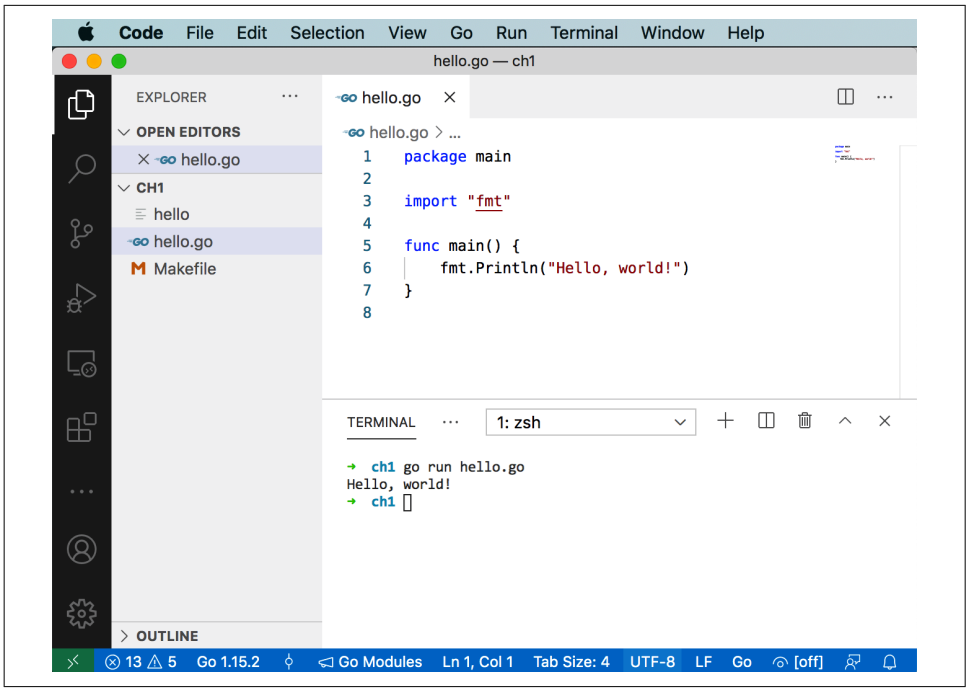

Figure 1-1. Visual Studio Code

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 download any additional tools to get it to work.

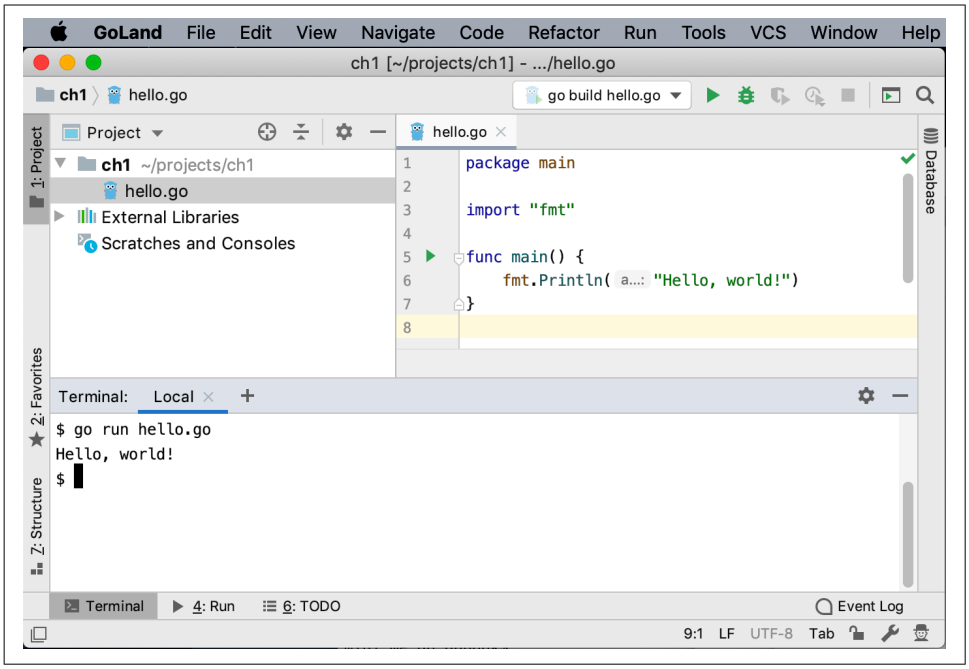

Figure 1-2. GoLand

If you already subscribe to IntelliJ Ultimate (or qualify for a free license), you can add Go support via a plug-in. Otherwise, you have to pay for GoLand; there is no free version available.

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 checking the Imports checkbox cleans up your imports like `goimports`. 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 been persistent for a long time, but I wouldn't use the playground as your source code repository).

Figure 1-3. The Go Playground

As you can see in Figure 1-4, you can even simulate multiple files by separating each file with a line that looks like `-- filename.go --`.

Be aware that The Go Playground is someone else's computer (in particular, it is Google's computer), so you don't have completely free rein. It always runs the latest stable version of Go. You cannot make network connections, and processes that run for too long or use too much memory are stopped. If your program depends on time, be aware that the clock is set to November 10, 2009, 23:00:00 UTC (the date of the initial announcement of Go). But 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.

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.

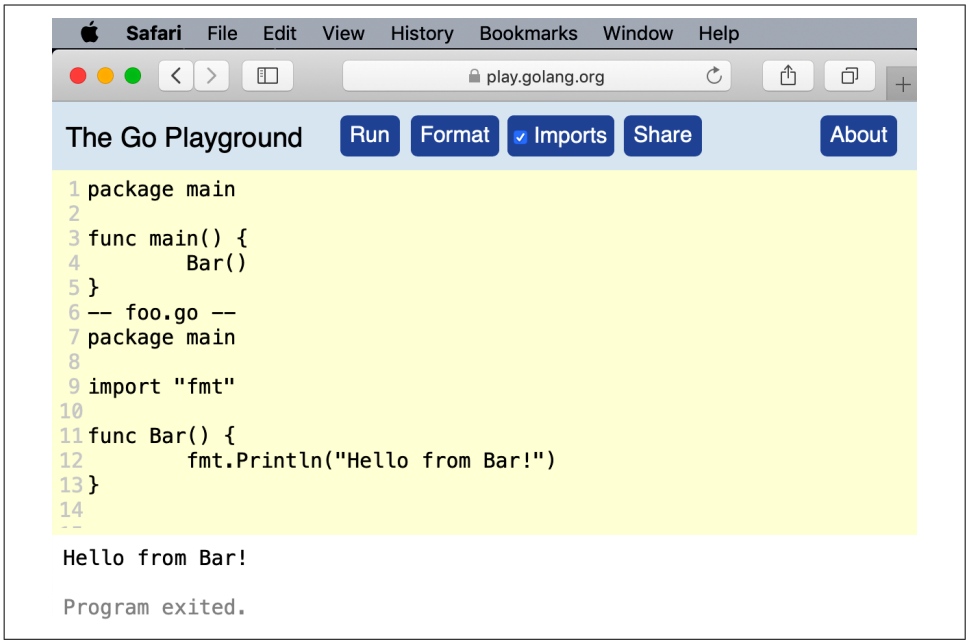

Figure 1-4. The Go Playground supports multiple files

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. This avoids the age-old developer excuse of "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. You may not be familiar with `make`, but it's been used to build programs on Unix systems since 1976.

Here's a sample Makefile to add to our very simple project:

```
.DEFAULT_GOAL := build

fmt:
	go fmt ./...
.PHONY:fmt

lint: fmt
	golint ./...
.PHONY:lint

vet: fmt
	go vet ./...
.PHONY:vet
```

```
build: vet
	go build hello.go
.PHONY: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 you ever create a directory in your project with the same name as a target.)

Once the Makefile is in the *ch1* directory, type:

```
make
```

You should see the following output:

```
go fmt ./...
go vet ./...
go build hello.go
```

By entering a single command, we make sure the code was formatted correctly, vet the code for nonobvious errors, and compile. We can also run the linter with `make lint`, 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`.)

Staying Up to Date

As with all programming languages, there are periodic updates to the Go development tools. Go programs are native binaries that don't rely on a separate runtime, 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.

Since Go 1.2, there has been a new major release roughly every six months. There are also minor 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. However, there might be (and have been) backward-incompatible changes to the flags or functionality of the `go` commands.

Despite these backward compatibility guarantees, bugs do happen, so it's natural to want to make sure that a new release doesn't break your programs. One option is to install a secondary Go environment. For example, if you are currently running version 1.15.2 and wanted to try out version 1.15.6, you would use the following commands:

```
$ go get golang.org/dl/go.1.15.6
$ go1.15.6 download
```

You can then use the command `go1.15.6` instead of the `go` command to see if version 1.15.6 works for your programs:

```
$ go1.15.6 build
```

Once you have validated that your code works, you can delete the secondary environment by finding its `GOROOT`, deleting it, and then deleting its binary from your *$GOPATH/bin* directory. Here's how to do that on Mac OS, Linux, and BSD:

```
$ go1.15.6 env GOROOT
/Users/gobook/sdk/go1.15.6
$ rm -rf $(go1.15.6 env GOROOT)
$ rm $(go env GOPATH)/bin/go1.15.6
```

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, expand the new version, and then delete the old version:

```
$ mv /usr/local/go /usr/local/old-go
$ tar -C /usr/local -xzf go1.15.2.linux-amd64.tar.gz
$ rm -rf /usr/local/old-go
```

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

Primitive Types and Declarations

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 primitive types and variables. While every programmer has experience with these concepts, Go does some things differently, and there are subtle differences between Go and other languages.

Built-in Types

Go has many of the same built-in types as 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 *literal* in Go refers to writing out a number, character, or string. There are four common kinds of literals that you'll find in Go programs. (There's a rare fifth kind of literal that we'll cover when discussing complex numbers.)

Integer literals are sequences of numbers; they are normally base ten, 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.

Floating point literals have decimal points 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. Like integer literals, you can use underscores to format your floating point literals.

Rune literals represent characters and are 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 (`'\''`), double quote (`'\"'`), and backslash (`'\\'`).

Practically speaking, use base ten to represent your number literals and, unless the context makes your code clearer, try to avoid using any of the hexadecimal escapes for rune 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.

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, in any of the forms allowed. The only characters that cannot appear 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, use a *raw string literal*. These are delimited with backquotes (`) and can contain any literal character except a backquote. When using a raw string literal, we write our multiline greeting like so:

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

As we'll see in "Explicit Type Conversion" on page 26 you can't even add two integer variables together if they are declared to be of different sizes. However, Go lets you use an integer literal in floating point expressions or even assign an integer literal to a floating point variable. This is because literals in Go are untyped; they can interact with any variable that's compatible with the literal. When we look at user-defined types in Chapter 7, we'll see that we can even use literals with user-defined types based on primitive 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.

Literals are untyped because Go is a practical language. It makes sense to avoid forcing a type until the developer specifies one. There are 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`.

As you will see in the section on variable assignment, 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 built-in 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 :=" on page 27.

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 four 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 65536
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" on page 26 for more details). Integer literals default to being of `int` type.

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" on page 26 and `uintptr` in Chapter 14.

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, write a pair of functions, one with `int64` for the parameters and variables and the other with `uint64`. (We talk more about functions and their parameters in Chapter 5.)

The reason why `int64` and `uint64` are the idiomatic choice in this situation is that Go doesn't have generics (yet) and doesn't have function overloading. Without these features, you'd need to write many functions with slightly different names to implement your algorithm. Using `int64` and `uint64` means that you can write the code once and let your callers use type conversions to pass values in and convert data that's returned.

You can see this pattern in the Go standard library with the functions `FormatInt`/`FormatUint` and `ParseInt`/`ParseUint` in the `strconv` package. There are other situations, like in the `math/bits` package, where the size of the integer matters. In those cases, you need to write a separate function for every integer type.

- In all other cases, just use `int`.

Unless you *need* to be explicit about the size or sign of an integer for performance or integration purposes, use the `int` type. Consider any other type to be a premature optimization until proven otherwise.

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" on page 174).

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	Smallest (nonzero) absolute value
float32	3.402823466385288598117041834845169254 40e+38	1.401298464324817070923729583289916131280e-45
float64	1.797693134862315708145274237317043567981e+308	4.940656458412465441765687928682213723651e-324

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. (We'll learn all about testing and profiling in Chapter 13.)

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.

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!

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. For example, if you store the number –3.1415 in a `float64`, the 64-bit representation in memory looks like:

```
1100000000001001001000011100101011000000100000110001001001101111
```

which is exactly equal to –3.14150000000000018118839761883.

Many programmers learn at some point how integers are represented in binary (rightmost position is 1, next is 2, next is 4, and so on). Floating point numbers are very different. Out of the 64 bits above, one is used to represent the sign (positive or negative), 11 are used to represent a base two exponent, and 52 bits are used to represent the number in a normalized format (called the *mantissa*).

You can learn more about IEEE 754 on its Wikipedia page.

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.

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 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 built-in 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.

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.

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 z float64 = float64(x) + y
var d int = x + int(y)
```

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 `z`, we convert `x` to a `float64` using a `float64` type conversion, and for `d`, we convert `y` to an `int` using an `int` type conversion. When you run this code, it prints out 40.2 40.

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 == ""`.

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.

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 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 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 Variables" on page 62 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 and "The comma ok Idiom" on page 54).

You should rarely declare variables outside of functions, in what's called the *package block* (see "Blocks" on page 61). 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.

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

When developers learn a new programming language, they try to map familiar concepts. Many languages have a way to declare a value is 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 2-3 on The Go Playground.

Example 2-3. `const` declarations

```
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
    y = "bye"

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

If you try to run this code, compilations fails with the following error messages:

```
./const.go:20:4: cannot assign to x
./const.go:21:4: cannot assign to y
```

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.

However, `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

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. 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 "Go Is Call By Value" on page 104, we'll see how Go prevents modifications to variables that are passed as parameters to functions.

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 "iota Is for Enumerations—Sometimes" on page 137.

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
    x = 20
    fmt.Println(x)
    x = 30
}
```

While the compiler and `go vet` do not catch the unused assignments of 10 and 30 to `x`, `golangci-lint` detects them:

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

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 2-4 perfectly valid Go.

Example 2-4. Variable names you should never use

```
_0 := 0_0
_𝟙 := 20
ո := 3
 a  := "hello" // Unicode U+FF41
fmt.Println(_0)
fmt.Println(_𝟙)
fmt.Println(ո)
fmt.Println( a )
```

While this code works, *do not* name your variables 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 2-5 on The Go Playground.

Example 2-5. Using look-alike code points for variable names

```
func main() {
     a  := "hello"   // Unicode U+FF41
    a := "goodbye" // standard lowercase a (Unicode U+0061)
    fmt.Println( a )
    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, Go uses camel case (names like indexCounter or numberTries) when an identifier name consists of multiple words.

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 9.

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. 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; we will mention them 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, `f` for floats, `b` for booleans). 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.

Wrapping Up

We've covered a lot of ground here, understanding how to use the built-in 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 learn about encodings.

CHAPTER 3

Composite Types

In the last chapter, we looked at simple types: numbers, booleans, and strings. In this chapter, we'll learn about the composite types in Go, the built-in functions that support them, and the best practices for working with them.

Arrays—Too Rigid to Use Directly

Like most programming languages, Go has arrays. However, arrays are rarely used directly in Go. We'll learn why in a bit, but first we'll quickly cover array declaration syntax and use.

All of the elements in the array must be of the type that's specified (this doesn't mean they are always of the same type). There are a few different declaration styles. In the first, you specify the size of the array and the type of the elements in the array:

```
var x [3]int
```

This creates an array of three `int`s. Since no values were specified, all of the positions (`x[0]`, `x[1]`, and `x[2]`) are initialized to the zero value for an `int`, which is (of course) 0. If you have initial values for the array, you specify them with an *array literal*:

```
var x = [3]int{10, 20, 30}
```

If you have a *sparse array* (an array where most elements are set to their zero value), you can specify only the indices with values in the array literal:

```
var x = [12]int{1, 5: 4, 6, 10: 100, 15}
```

This creates an array of 12 `int`s with the following values: [1, 0, 0, 0, 0, 4, 6, 0, 0, 0, 100, 15].

When using an array literal to initialize an array, you can leave off the number and use ... instead:

```
var x = [...]int{10, 20, 30}
```

You can use == and != to compare arrays:

```
var x = [...]int{1, 2, 3}
var y = [3]int{1, 2, 3}
fmt.Println(x == y) // prints true
```

Go only has one-dimensional arrays, but you can simulate multidimensional arrays:

```
var x [2][3]int
```

This declares `x` to be an array of length 2 whose type is an array of `int`s of length 3. This sounds pedantic, but there are languages with true matrix support; Go isn't one of them.

Like most languages, arrays in Go are read and written using bracket syntax:

```
x[0] = 10
fmt.Println(x[2])
```

You cannot read or write past the end of an array or use a negative index. If you do this with a constant or literal index, it is a compile-time error. An out-of-bounds read or write with a variable index compiles but fails at runtime with a *panic* (we'll talk more about panics in "panic and recover" on page 174).

Finally, the built-in function `len` takes in an array and returns its length:

```
fmt.Println(len(x))
```

Earlier I said that arrays in Go are rarely used explicitly. This is because they come with an unusual limitation: Go considers the *size* of the array to be part of the *type* of the array. This makes an array that's declared to be `[3]int` a different type from an array that's declared to be `[4]int`. This also means that you cannot use a variable to specify the size of an array, because types must be resolved at compile time, not at runtime.

What's more, *you can't use a type conversion to convert arrays of different sizes to identical types*. Because you can't convert arrays of different sizes into each other, you can't write a function that works with arrays of any size and you can't assign arrays of different sizes to the same variable.

We'll learn how arrays work behind the scenes when we discuss memory layout in Chapter 6.

Due to these restrictions, don't use arrays unless you know the exact length you need ahead of time. For example, some of the cryptographic functions in the standard library return arrays because the sizes of checksums are defined as part of the algorithm. This is the exception, not the rule.

This raises the question: why is such a limited feature in the language? The main reason why arrays exist in Go is to provide the backing store for *slices*, which are one of the most useful features of Go.

Slices

Most of the time, when you want a data structure that holds a sequence of values, a slice is what you should use. What makes slices so useful is that the length is *not* part of the type for a slice. This removes the limitations of arrays. We can write a single function that processes slices of any size (we'll cover function writing in Chapter 5), and we can grow slices as needed. After going over the basics of using slices in Go, we'll cover the best ways to use them.

Working with slices looks quite a bit like working with arrays, but there are subtle differences. The first thing to notice is that we don't specify the size of the slice when we declare it:

```
var x = []int{10, 20, 30}
```

Using [...] makes an array. Using [] makes a slice.

This creates a slice of 3 `int`s using a *slice literal*. Just like arrays, we can also specify only the indices with values in the slice literal:

```
var x = []int{1, 5: 4, 6, 10: 100, 15}
```

This creates a slice of 12 `int`s with the following values: [1, 0, 0, 0, 0, 4, 6, 0, 0, 0, 100, 15].

You can simulate multidimensional slices and make a slice of slices:

```
var x [][]int
```

You read and write slices using bracket syntax, and, just like with arrays, you can't read or write past the end or use a negative index:

```
x[0] = 10
fmt.Println(x[2])
```

So far, slices have seemed identical to arrays. We start to see the differences between arrays and slices when we look at declaring slices without using a literal:

```
var x []int
```

This creates a slice of `int`s. Since no value is assigned, `x` is assigned the zero value for a slice, which is something we haven't seen before: `nil`. We'll talk more about `nil` in Chapter 6, but it is slightly different from the `null` that's found in other languages. In Go, `nil` is an identifier that represents the lack of a value for some types. Like the untyped numeric constants we saw in the previous chapter, `nil` has no type, so it can be assigned or compared against values of different types. A `nil` slice contains nothing.

A slice is the first type we've seen that isn't *comparable*. It is a compile-time error to use `==` to see if two slices are identical or `!=` to see if they are different. The only thing you can compare a slice with is `nil`:

```
fmt.Println(x == nil) // prints true
```

The `reflect` package contains a function called `DeepEqual` that can compare almost anything, including slices. It's primarily intended for testing, but you could use it to compare slices if you needed to. We'll look at it when we discuss reflection in Chapter 14.

len

Go provides several built-in functions to work with its built-in types. We already saw the `complex`, `real`, and `imag` built-in functions to build and take apart complex numbers. There are several built-in functions for slices, too. We've already seen the built-in `len` function when looking at arrays. It works for slices, too, and when you pass a `nil` slice to `len`, it returns 0.

Functions like `len` are built in to Go because they can do things that can't be done by the functions that you can write. We've already seen that `len`'s parameter can be any type of array or any type of slice. We'll soon see that it also works for strings and maps. In "Channels" on page 206, we'll see it working with channels. Trying to pass a variable of any other type to `len` is a compile-time error. As we'll see in Chapter 5, Go doesn't let developers write functions that behave this way.

append

The built-in `append` function is used to grow slices:

```
var x []int
x = append(x, 10)
```

The `append` function takes at least two parameters, a slice of any type and a value of that type. It returns a slice of the same type. The returned slice is assigned back to the slice that's passed in. In this example, we are appending to a `nil` slice, but you can append to a slice that already has elements:

```
var x = []int{1, 2, 3}
x = append(x, 4)
```

You can append more than one value at a time:

```
x = append(x, 5, 6, 7)
```

One slice is appended onto another by using the … operator to expand the source slice into individual values (we'll learn more about the … operator in "Variadic Input Parameters and Slices" on page 89):

```
y := []int{20, 30, 40}
x = append(x, y...)
```

It is a compile-time error if you forget to assign the value returned from `append`. You might be wondering why as it seems a bit repetitive. We will talk about this in greater detail in Chapter 5, but Go is a *call by value* language. Every time you pass a parameter to a function, Go makes a copy of the value that's passed in. Passing a slice to the `append` function actually passes a copy of the slice to the function. The function adds the values to the copy of the slice and returns the copy. You then assign the returned slice back to the variable in the calling function.

Capacity

As we've seen, a slice is a sequence of values. Each element in a slice is assigned to consecutive memory locations, which makes it quick to read or write these values. Every slice has a *capacity*, which is the number of consecutive memory locations reserved. This can be larger than the length. Each time you append to a slice, one or more values is added to the end of the slice. Each value added increases the length by one. When the length reaches the capacity, there's no more room to put values. If you try to add additional values when the length equals the capacity, the `append` function uses the Go runtime to allocate a new slice with a larger capacity. The values in the original slice are copied to the new slice, the new values are added to the end, and the new slice is returned.

The Go Runtime

Every high-level language relies on a set of libraries to enable programs written in that language to run, and Go is no exception. The Go runtime provides services like memory allocation and garbage collection, concurrency support, networking, and implementations of built-in types and functions.

The Go runtime is compiled into every Go binary. This is different from languages that use a virtual machine, which must be installed separately to allow programs written in those languages to function. Including the runtime in the binary makes it easier to distribute Go programs and avoids worries about compatibility issues between the runtime and the program.

When a slice grows via `append`, it takes time for the Go runtime to allocate new memory and copy the existing data from the old memory to the new. The old memory also needs to be garbage collected. For this reason, the Go runtime usually increases a slice by more than one each time it runs out of capacity. The rules as of Go 1.14 are to double the size of the slice when the capacity is less than 1,024 and then grow by at least 25% afterward.

Just as the built-in `len` function returns the current length of a slice, the built-in `cap` function returns the current capacity of a slice. It is used far less frequently than `len`. Most of the time, `cap` is used to check if a slice is large enough to hold new data, or if a call to `make` is needed to create a new slice.

You can also pass an array to the `cap` function, but `cap` always returns the same value as `len` for arrays. Don't put it in your code, but save this trick for Go trivia night.

Let's take a look at how adding elements to a slice changes the length and capacity. Run the code in Example 3-1 on The Go Playground or on your machine.

Example 3-1. Understanding capacity

```
var x []int
fmt.Println(x, len(x), cap(x))
x = append(x, 10)
fmt.Println(x, len(x), cap(x))
x = append(x, 20)
fmt.Println(x, len(x), cap(x))
x = append(x, 30)
fmt.Println(x, len(x), cap(x))
x = append(x, 40)
fmt.Println(x, len(x), cap(x))
x = append(x, 50)
fmt.Println(x, len(x), cap(x))
```

When you build and run the code, you'll see the following output. Notice how and when the capacity increases:

```
[] 0 0
[10] 1 1
[10 20] 2 2
[10 20 30] 3 4
[10 20 30 40] 4 4
[10 20 30 40 50] 5 8
```

While it's nice that slices grow automatically, it's far more efficient to size them once. If you know how many things you plan to put into a slice, create the slice with the correct initial capacity. We do that with the `make` function.

make

We've already seen two ways to declare a slice, using a slice literal or the `nil` zero value. While useful, neither way allows you to create an empty slice that already has a length or capacity specified. That's the job of the built-in `make` function. It allows us to specify the type, length, and, optionally, the capacity. Let's take a look:

```
x := make([]int, 5)
```

This creates an `int` slice with a length of 5 and a capacity of 5. Since it has a length of 5, `x[0]` through `x[4]` are valid elements, and they are all initialized to 0.

One common beginner mistake is to try to populate those initial elements using `append`:

```
x := make([]int, 5)
x = append(x, 10)
```

The 10 is placed at the end of the slice, *after* the zero values in positions 0–4 because `append` always increases the length of a slice. The value of `x` is now [0 0 0 0 0 10], with a length of 6 and a capacity of 10 (the capacity was doubled as soon as the sixth element was appended).

We can also specify an initial capacity with `make`:

```
x := make([]int, 5, 10)
```

This creates an `int` slice with a length of 5 and a capacity of 10.

You can also create a slice with zero length, but a capacity that's greater than zero:

```
x := make([]int, 0, 10)
```

In this case, we have a non-nil slice with a length of 0, but a capacity of 10. Since the length is 0, we can't directly index into it, but we can append values to it:

```
x := make([]int, 0, 10)
x = append(x, 5,6,7,8)
```

The value of x is now [5 6 7 8], with a length of 4 and a capacity of 10.

Never specify a capacity that's less than the length! It is a compile-time error to do so with a constant or numeric literal. If you use a variable to specify a capacity that's smaller than the length, your program will panic at runtime.

Declaring Your Slice

Now that we've seen all these different ways to create slices, how do you choose which slice declaration style to use? The primary goal is to minimize the number of times the slice needs to grow. If it's possible that the slice won't need to grow at all (because your function might return nothing), use a `var` declaration with no assigned value to create a `nil` slice, as shown in Example 3-2.

Example 3-2. Declaring a slice that might stay nil

```
var data []int
```

You can create a slice using an empty slice literal:

```
var x = []int{}
```

This creates a zero-length slice, which is non-nil (comparing it to `nil` returns `false`). Otherwise, a `nil` slice works identically to a zero-length slice. The only situation where a zero-length slice is useful is when converting a slice to JSON. We'll look at this more in "encoding/json" on page 241.

If you have some starting values, or if a slice's values aren't going to change, then a slice literal is a good choice (see Example 3-3).

Example 3-3. Declaring a slice with default values

```
data := []int{2, 4, 6, 8} // numbers we appreciate
```

If you have a good idea of how large your slice needs to be, but don't know what those values will be when you are writing the program, use `make`. The question then becomes whether you should specify a nonzero length in the call to `make` or specify a zero length and a nonzero capacity. There are three possibilities:

- If you are using a slice as a buffer (we'll see this in "io and Friends" on page 233), then specify a nonzero length.
- If you are *sure* you know the exact size you want, you can specify the length and index into the slice to set the values. This is often done when transforming values in one slice and storing them in a second. The downside to this approach is that if you have the size wrong, you'll end up with either zero values at the end of the slice or a panic from trying to access elements that don't exist.
- In other situations, use `make` with a zero length and a specified capacity. This allows you to use `append` to add items to the slice. If the number of items turns out to be smaller, you won't have an extraneous zero value at the end. If the number of items is larger, your code will not panic.

The Go community is split between the second and third approaches. I personally prefer using `append` with a slice initialized to a zero length. It might be slower in some situations, but it is less likely to introduce a bug.

`append` always increases the length of a slice! If you have specified a slice's length using `make`, be sure that you mean to append to it before you do so, or you might end up with a bunch of surprise zero values at the beginning of your slice.

Slicing Slices

A *slice expression* creates a slice from a slice. It's written inside brackets and consists of a starting offset and an ending offset, separated by a colon (:). If you leave off the starting offset, 0 is assumed. Likewise, if you leave off the ending offset, the end of the slice is substituted. You can see how this works by running the code in Example 3-4 on The Go Playground.

Example 3-4. Slicing slices

```
x := []int{1, 2, 3, 4}
y := x[:2]
z := x[1:]
d := x[1:3]
e := x[:]
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
fmt.Println("d:", d)
fmt.Println("e:", e)
```

It gives the following output:

```
x: [1 2 3 4]
y: [1 2]
z: [2 3 4]
d: [2 3]
e: [1 2 3 4]
```

Slices share storage sometimes

When you take a slice from a slice, you are *not* making a copy of the data. Instead, you now have two variables that are sharing memory. This means that changes to an element in a slice affect all slices that share that element. Let's see what happens when we change values. You can run the code in Example 3-5 on The Go Playground.

Example 3-5. Slices with overlapping storage

```
x := []int{1, 2, 3, 4}
y := x[:2]
z := x[1:]
x[1] = 20
y[0] = 10
z[1] = 30
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

You get the following output:

```
x: [10 20 30 4]
y: [10 20]
z: [20 30 4]
```

Changing `x` modified both `y` and `z`, while changes to `y` and `z` modified `x`.

Slicing slices gets extra confusing when combined with `append`. Try out the code in Example 3-6 on The Go Playground.

Example 3-6. `append` makes overlapping slices more confusing

```
x := []int{1, 2, 3, 4}
y := x[:2]
fmt.Println(cap(x), cap(y))
y = append(y, 30)
fmt.Println("x:", x)
fmt.Println("y:", y)
```

Running this code gives the following output:

```
4 4
x: [1 2 30 4]
y: [1 2 30]
```

What's going on? Whenever you take a slice from another slice, the subslice's capacity is set to the capacity of the original slice, minus the offset of the subslice within the original slice. This means that any unused capacity in the original slice is also shared with any subslices.

When we make the `y` slice from `x`, the length is set to 2, but the capacity is set to 4, the same as `x`. Since the capacity is 4, appending onto the end of `y` puts the value in the third position of `x`.

This behavior creates some very odd scenarios, with multiple slices appending and overwriting each other's data. See if you can guess what the code in Example 3-7 prints out, then run it on The Go Playground to see if you guessed correctly.

Example 3-7. Even more confusing slices

```
x := make([]int, 0, 5)
x = append(x, 1, 2, 3, 4)
y := x[:2]
z := x[2:]
fmt.Println(cap(x), cap(y), cap(z))
y = append(y, 30, 40, 50)
x = append(x, 60)
z = append(z, 70)
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

To avoid complicated slice situations, you should either never use `append` with a subslice or make sure that `append` doesn't cause an overwrite by using a *full slice expression*. This is a little weird, but it makes clear how much memory is shared between the parent slice and the subslice. The full slice expression includes a third part, which indicates the last position in the parent slice's capacity that's available for the subslice. Subtract the starting offset from this number to get the subslice's capacity. Example 3-8 shows lines three and four from the previous example, modified to use full slice expressions.

Example 3-8. The full slice expression protects against `append`

```
y := x[:2:2]
z := x[2:4:4]
```

You can try out this code on The Go Playground. Both `y` and `z` have a capacity of 2. Because we limited the capacity of the subslices to their lengths, appending additional elements onto `y` and `z` created new slices that didn't interact with the other slices. After this code runs, `x` is set to [1 2 3 4 60], `y` is set to [1 2 30 40 50], and `z` is set to [3 4 70].

Be very careful when taking a slice of a slice! Both slices share the same memory and changes to one are reflected in the other. Avoid modifying slices after they have been sliced or if they were produced by slicing. Use a three-part slice expression to prevent `append` from sharing capacity between slices.

Converting Arrays to Slices

Slices aren't the only thing you can slice. If you have an array, you can take a slice from it using a slice expression. This is a useful way to bridge an array to a function that only takes slices. However, be aware that taking a slice from an array has the same memory-sharing properties as taking a slice from a slice. If you run the following code on The Go Playground:

```
x := [4]int{5, 6, 7, 8}
y := x[:2]
z := x[2:]
x[0] = 10
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

you get the output:

```
x: [10 6 7 8]
y: [10 6]
z: [7 8]
```

copy

If you need to create a slice that's independent of the original, use the built-in `copy` function. Let's take a look at a simple example, which you can run on The Go Playground:

```
x := []int{1, 2, 3, 4}
y := make([]int, 4)
num := copy(y, x)
fmt.Println(y, num)
```

You get the output:

```
[1 2 3 4] 4
```

The `copy` function takes two parameters. The first is the destination slice and the second is the source slice. It copies as many values as it can from source to destination, limited by whichever slice is smaller, and returns the number of elements copied. The *capacity* of `x` and `y` doesn't matter; it's the length that's important.

You don't need to copy an entire slice. The following code copies the first two elements of a four-element slice into a two-element slice:

```
x := []int{1, 2, 3, 4}
y := make([]int, 2)
num = copy(y, x)
```

The variable `y` is set to [1 2] and `num` is set to 2.

You could also copy from the middle of the source slice:

```
x := []int{1, 2, 3, 4}
y := make([]int, 2)
copy(y, x[2:])
```

We are copying the third and fourth elements in `x` by taking a slice of the slice. Also note that we don't assign the output of `copy` to a variable. If you don't need the number of elements copied, you don't need to assign it.

The `copy` function allows you to copy between two slices that cover overlapping sections of an underlying slice:

```
x := []int{1, 2, 3, 4}
num = copy(x[:3], x[1:])
fmt.Println(x, num)
```

In this case, we are copying the last three values in `x` on top of the first three values of `x`. This prints out [2 3 4 4] 3.

You can use `copy` with arrays by taking a slice of the array. You can make the array either the source or the destination of the copy. You can try out the following code on The Go Playground:

```
x := []int{1, 2, 3, 4}
d := [4]int{5, 6, 7, 8}
y := make([]int, 2)
copy(y, d[:])
fmt.Println(y)
copy(d[:], x)
fmt.Println(d)
```

The first call to `copy` copies the first two values in array `d` into slice `y`. The second copies all of the values in slice `x` into array `d`. This produces the output:

```
[5 6]
[1 2 3 4]
```

Strings and Runes and Bytes

Now that we've talked about slices, we can go back and look at strings again. You might think that a string in Go is made out of runes, but that's not the case. Under the covers, Go uses a sequence of bytes to represent a string. These bytes don't have to be in any particular character encoding, but several Go library functions (and the `for-range` loop that we discuss in the next chapter) assume that a string is composed of a sequence of UTF-8-encoded code points.

According to the language specification, Go source code is always written in UTF-8. Unless you use hexadecimal escapes in a string literal, your string literals are written in UTF-8.

Just like you can extract a single value from an array or a slice, you can extract a single value from a string by using an *index expression*:

```
var s string = "Hello there"
var b byte = s[6]
```

Like arrays and slices, string indexes are zero-based; in this example, `b` is assigned the value of the seventh value in `s`, which is `t`.

The slice expression notation that we used with arrays and slices also works with strings:

```
var s string = "Hello there"
var s2 string = s[4:7]
var s3 string = s[:5]
var s4 string = s[6:]
```

This assigns "o t" to `s2`, "Hello" to `s3`, and "there" to `s4`.

While it's handy that Go allows us to use slicing notation to make substrings and use index notation to extract individual entries from a string, you should be very careful when doing so. Since strings are immutable, they don't have the modification problems that slices of slices do. There is a different problem, though. A string is composed of a sequence of bytes, while a code point in UTF-8 can be anywhere from one to four bytes long. Our previous example was entirely composed of code points that are one byte long in UTF-8, so everything worked out as expected. But when dealing with languages other than English or with emojis, you run into code points that are multiple bytes long in UTF-8:

```
var s string = "Hello 🌞"
var s2 string = s[4:7]
var s3 string = s[:5]
var s4 string = s[6:]
```

In this example, `s3` will still be equal to "Hello." The variable `s4` is set to the sun emoji. But `s2` is not set to "o ☼." Instead, you get "o  ." That's because we only copied the first byte of the sun emoji's code point, which is invalid.

Go allows you to pass a string to the built-in `len` function to find the length of the string. Given that string index and slice expressions count positions in bytes, it's not surprising that the length returned is the length in bytes, not in code points:

```
var s string = "Hello ☼"
fmt.Println(len(s))
```

This code prints out 10, not 7, because it takes four bytes to represent the sun with smiling face emoji in UTF-8.

Even though Go allows you to use slicing and indexing syntax with strings, you should only use it when you know that your string only contains characters that take up one byte.

Because of this complicated relationship between runes, strings, and bytes, Go has some interesting type conversions between these types. A single rune or byte can be converted to a string:

```
var a rune    = 'x'
var s string  = string(a)
var b byte    = 'y'
var s2 string = string(b)
```

A common bug for new Go developers is to try to make an `int` into a `string` by using a type conversion:

```
var x int = 65
var y = string(x)
fmt.Println(y)
```

This results in `y` having the value "A," not "65." As of Go 1.15, `go vet` blocks a type conversion to string from any integer type other than `rune` or `byte`.

A string can be converted back and forth to a slice of bytes or a slice of runes. Try Example 3-9 out on The Go Playground.

Example 3-9. Converting strings to slices

```
var s string = "Hello, ☼"
var bs []byte = []byte(s)
var rs []rune = []rune(s)
```

```
fmt.Println(bs)
fmt.Println(rs)
```

When you run this code, you see:

```
[72 101 108 108 111 44 32 240 159 140 158]
[72 101 108 108 111 44 32 127774]
```

The first output line has the string converted to UTF-8 bytes. The second has the string converted to runes.

Most data in Go is read and written as a sequence of bytes, so the most common string type conversions are back and forth with a slice of bytes. Slices of runes are uncommon.

UTF-8

UTF-8 is the most commonly used encoding for Unicode. Unicode uses four bytes (32 bits) to represent each *code point*, the technical name for each character and modifier. Given this, the simplest way to represent Unicode code points is to store four bytes for each code point. This is called UTF-32. It is mostly unused because it wastes so much space. Due to Unicode implementation details, 11 of the 32 bits are always zero. Another common encoding is UTF-16, which uses one or two 16-bit (2-byte) sequences to represent each code point. This is also wasteful; much of the content in the world is written using code points that fit into a single byte. And that's where UTF-8 comes in.

UTF-8 is very clever. It lets you use a single byte to represent the Unicode characters whose values are below 128 (which includes all of the letters, numbers, and punctuation commonly used in English), but expands to a maximum of four bytes to represent Unicode code points with larger values. The result is that the *worst* case for UTF-8 is the same as using UTF-32. UTF-8 has some other nice properties. Unlike UTF-32 and UTF-16, you don't have to worry about little-endian versus big-endian. It also allows you to look at any byte in a sequence and tell if you are at the start of a UTF-8 sequence, or somewhere in the middle. That means you can't accidentally read a character incorrectly.

The only downside is that you cannot randomly access a string encoded with UTF-8. While you can detect if you are in the middle of a character, you can't tell how many characters in you are. You need to start at the beginning of the string and count. Go doesn't require a string to be written in UTF-8, but it strongly encourages it. We'll see how to work with UTF-8 strings in upcoming chapters.

Fun fact: UTF-8 was invented in 1992 by Ken Thompson and Rob Pike, two of the creators of Go.

Rather than use the slice and index expressions with strings, you should extract substrings and code points from strings using the functions in the `strings` and `unicode/utf8` packages in the standard library. In the next chapter, we'll see how to use a `for-range` loop to iterate over the code points in a string.

Maps

Slices are useful when you have sequential data. Like most languages, Go provides a built-in data type for situations where you want to associate one value to another. The map type is written as `map[keyType]valueType`. Let's take a look at a few ways to declare maps. First, you can use a `var` declaration to create a map variable that's set to its zero value:

```
var nilMap map[string]int
```

In this case, `nilMap` is declared to be a map with `string` keys and `int` values. The zero value for a map is `nil`. A `nil` map has a length of 0. Attempting to read a `nil` map always returns the zero value for the map's value type. However, attempting to write to a `nil` map variable causes a panic.

We can use a `:=` declaration to create a map variable by assigning it a *map literal*:

```
totalWins := map[string]int{}
```

In this case, we are using an empty map literal. This is not the same as a `nil` map. It has a length of 0, but you can read and write to a map assigned an empty map literal. Here's what a nonempty map literal looks like:

```
teams := map[string][]string {
    "Orcas": []string{"Fred", "Ralph", "Bijou"},
    "Lions": []string{"Sarah", "Peter", "Billie"},
    "Kittens": []string{"Waldo", "Raul", "Ze"},
}
```

A map literal's body is written as the key, followed by a colon (:), then the value. There's a comma separating each key-value pair in the map, even on the last line. In this example, the value is a slice of strings. The type of the value in a map can be anything. There are some restrictions on the types of the keys that we'll discuss in a bit.

If you know how many key-value pairs you intend to put in the map, but don't know the exact values, you can use `make` to create a map with a default size:

```
ages := make(map[int][]string, 10)
```

Maps created with `make` still have a length of 0, and they can grow past the initially specified size.

Maps are like slices in several ways:

- Maps automatically grow as you add key-value pairs to them.
- If you know how many key-value pairs you plan to insert into a map, you can use `make` to create a map with a specific initial size.
- Passing a map to the `len` function tells you the number of key-value pairs in a map.
- The zero value for a map is `nil`.
- Maps are not comparable. You can check if they are equal to `nil`, but you cannot check if two maps have identical keys and values using == or differ using `!=`.

The key for a map can be any comparable type. This means you cannot use a slice or a map as the key for a map.

When should you use a map and when should you use a slice? Slices are for lists of data, especially for data that's processed sequentially. Maps are useful when you have data that's organized according to a value that's not a strictly increasing order.

Use a map when the order of elements doesn't matter. Use a slice when the order of elements is important.

What Is a Hash Map?

In computer science, a *map* is a data structure that associates (or maps) one value to another. Maps can be implemented several ways, each with their own trade-offs. The map that's built-in to Go is a *hash map*. In case you aren't familiar with the concept, here is a really quick overview.

A hash map does fast lookups of values based on a key. Internally, it's implemented as an array. When you insert a key and value, the key is turned into a number using a *hash algorithm*. These numbers are not unique for each key. The hash algorithm can turn different keys into the same number. That number is then used as an index into the array. Each element in that array is called a *bucket*. The key-value pair is then stored in the bucket. If there is already an identical key in the bucket, the previous value is replaced with the new value.

Each bucket is also an array; it can hold more than one value. When two keys map to the same bucket, that's called a *collision*, and the keys and values for both are stored in the bucket.

A read from a hash map works in the same way. You take the key, run the hash algorithm to turn it into a number, find the associated bucket, and then iterate over all the keys in the bucket to see if one of them is equal to the supplied key. If one is found, the value is returned.

You don't want to have too many collisions, because the more collisions, the slower the hash map gets, as you have to iterate over all the keys that mapped to the same bucket to find the one that you want. Clever hash algorithms are designed to keep collisions to a minimum. If enough elements are added, hash maps resize to rebalance the buckets and allow more entries.

Hash maps are really useful, but building your own is hard to get right. If you'd like to learn more about how Go does it, watch this talk from GopherCon 2016, Inside the Map Implementation.

Go doesn't require (or even allow) you to define your own hash algorithm or equality definition. Instead, the Go runtime that's compiled into every Go program has code that implements hash algorithms for all types that are allowed to be keys.

Reading and Writing a Map

Let's look at a short program that declares, writes to, and reads from a map. You can run the program in Example 3-10 on The Go Playground.

Example 3-10. Using a map

```
totalWins := map[string]int{}
totalWins["Orcas"] = 1
totalWins["Lions"] = 2
fmt.Println(totalWins["Orcas"])
fmt.Println(totalWins["Kittens"])
totalWins["Kittens"]++
fmt.Println(totalWins["Kittens"])
totalWins["Lions"] = 3
fmt.Println(totalWins["Lions"])
```

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

```
1
0
1
3
```

We assign a value to a map key by putting the key within brackets and using `=` to specify the value, and we read the value assigned to a map key by putting the key within brackets. Note that you cannot use `:=` to assign a value to a map key.

When we try to read the value assigned to a map key that was never set, the map returns the zero value for the map's value type. In this case, the value type is an `int`, so we get back a 0. You can use the `++` operator to increment the numeric value for a map key. Because a map returns its zero value by default, this works even when there's no existing value associated with the key.

The comma ok Idiom

As we've seen, a map returns the zero value if you ask for the value associated with a key that's not in the map. This is handy when implementing things like the counter we saw earlier. However, you sometimes do need to find out if a key is in a map. Go provides the *comma ok idiom* to tell the difference between a key that's associated with a zero value and a key that's not in the map:

```
m := map[string]int{
    "hello": 5,
    "world": 0,
}
v, ok := m["hello"]
fmt.Println(v, ok)

v, ok = m["world"]
fmt.Println(v, ok)

v, ok = m["goodbye"]
fmt.Println(v, ok)
```

Rather than assign the result of a map read to a single variable, with the comma ok idiom you assign the results of a map read to two variables. The first gets the value associated with the key. The second value returned is a bool. It is usually named `ok`. If `ok` is `true`, the key is present in the map. If `ok` is `false`, the key is not present. In this example, the code prints out 5 true, 0 true, and 0 false.

The comma ok idiom is used in Go when we want to differentiate between reading a value and getting back the zero value. We'll see it again when we read from channels in Chapter 10 and when we use type assertions in Chapter 7.

Deleting from Maps

Key-value pairs are removed from a map via the built-in `delete` function:

```
m := map[string]int{
    "hello": 5,
    "world": 10,
}
delete(m, "hello")
```

The `delete` function takes a map and a key and then removes the key-value pair with the specified key. If the key isn't present in the map or if the map is `nil`, nothing happens. The `delete` function doesn't return a value.

Using Maps as Sets

Many languages include a *set* in their standard library. A set is a data type that ensures there is at most one of a value, but doesn't guarantee that the values are in any particular order. Checking to see if an element is in a set is fast, no matter how many elements are in the set. (Checking to see if an element is in a slice takes longer as you add more elements to the slice.)

Go doesn't include a set, but you can use a map to simulate some of its features. Use the key of the map for the type that you want to put into the set and use a `bool` for the value. The code in Example 3-11 demonstrates the concept. You can run it on The Go Playground.

Example 3-11. Using a map as a set

```
intSet := map[int]bool{}
vals := []int{5, 10, 2, 5, 8, 7, 3, 9, 1, 2, 10}
for _, v := range vals {
    intSet[v] = true
}
fmt.Println(len(vals), len(intSet))
fmt.Println(intSet[5])
fmt.Println(intSet[500])
if intSet[100] {
    fmt.Println("100 is in the set")
}
```

We want a set of `int`s, so we create a map where the keys are of `int` type and the values are of `bool` type. We iterate over the values in `vals` using a `for-range` loop (which we discuss in "The for-range Statement" on page 71) to place them into `intSet`, associating each `int` with the boolean value `true`.

We wrote 11 values into `intSet`, but the length of `intSet` is 8, because you cannot have duplicate keys in a map. If we look for 5 in `intSet`, it returns `true`, because we have a key with the value 5. However, if we look for 500 or 100 in `intSet`, it returns `false`. This is because we haven't put either value into `intSet`, which causes the map to return the zero value for the map value, and the zero value for a `bool` is `false`.

If you need sets that provide operations like union, intersection, and subtraction, you can either write one yourself or use one of the many third-party libraries that provide the functionality. (We'll learn more about using third-party libraries in Chapter 9.)

Some people prefer to use `struct{}` for the value when a map is being used to implement a set. (We'll discuss structs in the next section.) The advantage is that an empty struct uses zero bytes, while a boolean uses one byte.

The disadvantage is that using a `struct{}` makes your code more clumsy. You have a less obvious assignment, and you need to use the comma ok idiom to check if a value is in the set:

```
intSet := map[int]struct{}{}
vals := []int{5, 10, 2, 5, 8, 7, 3, 9, 1, 2, 10}
for _, v := range vals {
    intSet[v] = struct{}{}
}
if _, ok := intSet[5]; ok {
    fmt.Println("5 is in the set")
}
```

Unless you have very large sets, it is unlikely that the difference in memory usage is significant enough to outweigh the disadvantages.

Structs

Maps are a convenient way to store some kinds of data, but they have limitations. They don't define an API since there's no way to constrain a map to only allow certain keys. Also, all of the values in a map must be of the same type. For these reasons, maps are not an ideal way to pass data from function to function. When you have related data that you want to group together, you should define a *struct*.

If you already know an object-oriented language, you might be wondering about the difference between classes and structs. The difference is simple: Go doesn't have classes, because it doesn't have inheritance. This doesn't mean that Go doesn't have some of the features of object-oriented languages, it just does things a little differently. We'll learn more about the object-oriented features of Go in Chapter 7.

Most languages have a concept that's similar to a struct, and the syntax that Go uses to read and write structs should look familiar:

```
type person struct {
    name string
    age  int
    pet  string
}
```

A struct type is defined with the keyword `type`, the name of the struct type, the keyword `struct`, and a pair of braces ({}). Within the braces, you list the fields in the

struct. Just like we put the variable name first and the variable type second in a `var` declaration, we put the struct field name first and the struct field type second. Also note that unlike map literals, there are no commas separating the fields in a struct declaration. You can define a struct type inside or outside of a function. A struct type that's defined within a function can only be used within that function. (We'll learn more about functions in Chapter 5.)

Technically, you can scope a struct definition to any block level. We'll learn more about blocks in Chapter 4.

Once a struct type is declared, we can define variables of that type:

```
var fred person
```

Here we are using a `var` declaration. Since no value is assigned to `fred`, it gets the zero value for the `person` struct type. A zero value struct has every field set to the field's zero value.

A *struct literal* can be assigned to a variable as well:

```
bob := person{}
```

Unlike maps, there is no difference between assigning an empty struct literal and not assigning a value at all. Both initialize all of the fields in the struct to their zero values. There are two different styles for a nonempty struct literal. A struct literal can be specified as a comma-separated list of values for the fields inside of braces:

```
julia := person{
    "Julia",
    40,
    "cat",
}
```

When using this struct literal format, a value for every field in the struct must be specified, and the values are assigned to the fields in the order they were declared in the struct definition.

The second struct literal style looks like the map literal style:

```
beth := person{
    age:  30,
    name: "Beth",
}
```

You use the names of the fields in the struct to specify the values. When you use this style, you can leave out keys and specify the fields in any order. Any field not specified is set to its zero value. You cannot mix the two struct literal styles: either all of the

fields are specified with keys, or none of them are. For small structs where all fields are always specified, the simpler struct literal style is fine. In other cases, use the key names. It's more verbose, but it makes clear what value is being assigned to what field without having to reference the struct definition. It's also more maintainable. If you initialize a struct without using the field names and a future version of the struct adds additional fields, your code will no longer compile.

A field in a struct is accessed with dotted notation:

```
bob.name = "Bob"
fmt.Println(beth.name)
```

Just like we use brackets for both reading and writing to a map, we use dotted notation for reading and writing to struct fields.

Anonymous Structs

You can also declare that a variable implements a struct type without first giving the struct type a name. This is called an *anonymous struct*:

```
var person struct {
    name string
    age  int
    pet  string
}

person.name = "bob"
person.age = 50
person.pet = "dog"

pet := struct {
    name string
    kind string
}{
    name: "Fido",
    kind: "dog",
}
```

In this example, the types of the variables `person` and `pet` are anonymous structs. You assign (and read) fields in an anonymous struct just like you do for a named struct type. Just like you can initialize an instance of a named struct with a struct literal, you can do the same for an anonymous struct as well.

You might wonder when it's useful to have a data type that's only associated with a single instance. There are two common situations where anonymous structs are handy. The first is when you translate external data into a struct or a struct into external data (like JSON or protocol buffers). This is called *unmarshaling* and *marshaling* data. We'll learn how to do this in "encoding/json" on page 241.

Writing tests is another place where anonymous structs pop up. We'll use a slice of anonymous structs when writing table-driven tests in Chapter 13.

Comparing and Converting Structs

Whether or not a struct is comparable depends on the struct's fields. Structs that are entirely composed of comparable types are comparable; those with slice or map fields are not (as we will see in later chapters, function and channel fields also prevent a struct from being comparable).

Unlike in Python or Ruby, in Go there's no magic method that can be overridden to redefine equality and make `==` and `!=` work for incomparable structs. You can, of course, write your own function that you use to compare structs.

Just like Go doesn't allow comparisons between variables of different primitive types, Go doesn't allow comparisons between variables that represent structs of different types. Go does allow you to perform a type conversion from one struct type to another *if the fields of both structs have the same names, order, and types*. Let's see what this means. Given this struct:

```
type firstPerson struct {
    name string
    age  int
}
```

We can use a type conversion to convert an instance of `firstPerson` to `secondPerson`, but we can't use `==` to compare an instance of `firstPerson` and an instance of `secondPerson`, because they are different types:

```
type secondPerson struct {
    name string
    age  int
}
```

We can't convert an instance of `firstPerson` to `thirdPerson`, because the fields are in a different order:

```
type thirdPerson struct {
    age  int
    name string
}
```

We can't convert an instance of `firstPerson` to `fourthPerson` because the field names don't match:

```
type fourthPerson struct {
    firstName string
    age       int
}
```

Finally, we can't convert an instance of `firstPerson` to `fifthPerson` because there's an additional field:

```
type fifthPerson struct {
    name          string
    age           int
    favoriteColor string
}
```

Anonymous structs add a small twist to this: if two struct variables are being compared and at least one of them has a type that's an anonymous struct, you can compare them without a type conversion if the fields of both structs have the same names, order, and types. You can also assign between named and anonymous struct types if the fields of both structs have the same names, order, and types:

```
type firstPerson struct {
    name string
    age  int
}
f := firstPerson{
    name: "Bob",
    age:  50,
}
var g struct {
    name string
    age  int
}

// compiles -- can use = and == between identical named and anonymous structs
g = f
fmt.Println(f == g)
```

Wrapping Up

We've learned a lot about container types in Go. In addition to learning more about strings, we now know how to use the built-in generic container types, slices and maps. We can also construct our own composite types via structs. In our next chapter, we're going to take a look at Go's control structures, `for`, `if/else`, and `switch`. We will also learn how Go organizes code into blocks, and how the different block levels can lead to surprising behavior.

CHAPTER 4

Blocks, Shadows, and Control Structures

Now that we have covered variables, constants, and built-in types, we are ready to look at programming logic and organization. We'll start by explaining blocks and how they control when an identifier is available. Then we'll look at Go's control structures: `if`, `for`, and `switch`. Finally, we will talk about `goto` and the one situation when you should use it.

Blocks

Go lets you declare variables in lots of places. You can declare them outside of functions, as the parameters to functions, and as local variables within functions.

So far, we've only written the `main` function, but we'll write functions with parameters in the next chapter.

Each place where a declaration occurs is called a *block*. Variables, constants, types, and functions declared outside of any functions are placed in the *package* block. We've used `import` statements in our programs to gain access to printing and math functions (and will talk about them in detail in Chapter 9). They define names for other packages that are valid for the file that contains the `import` statement. These names are in the *file* block. All of the variables defined at the top level of a function (including the parameters to a function) are in a block. Within a function, every set of braces (`{}`) defines another block, and in a bit we will see that the control structures in Go define blocks of their own.

You can access an identifier defined in any outer block from within any inner block. This raises the question: what happens when you have a declaration with the same name as an identifier in a containing block? If you do that, you *shadow* the identifier created in the outer block.

Shadowing Variables

Before explaining what shadowing is, let's take a look at some code (see Example 4-1). You can run it on The Go Playground.

Example 4-1. Shadowing variables

```
func main() {
    x := 10
    if x > 5 {
        fmt.Println(x)
        x := 5
        fmt.Println(x)
    }
    fmt.Println(x)
}
```

Before you run this code, try to guess what it's going to print out:

- Nothing prints; the code does not compile
- 10 on line one, 5 on line two, 5 on line three
- 10 on line one, 5 on line two, 10 on line three

Here's what happens:

```
10
5
10
```

A shadowing variable is a variable that has the same name as a variable in a containing block. For as long as the shadowing variable exists, you cannot access a shadowed variable.

In this case, we almost certainly didn't want to create a brand-new `x` inside the `if` statement. Instead, we probably wanted to assign 5 to the `x` declared at the top level of the function block. At the first `fmt.Println` inside the `if` statement, we are able to access the `x` declared at the top level of the function. On the next line, though, we *shadow* `x` by *declaring a new variable with the same name* inside the block created by the `if` statement's body. At the second `fmt.Println`, when we access the variable named `x`, we get the shadowing variable, which has the value of 5. The closing brace for the `if` statement's body ends the block where the shadowing `x` exists, and at the

third `fmt.Println`, when we access the variable named `x`, we get the variable declared at the top level of the function, which has the value of 10. Notice that this `x` didn't disappear or get reassigned; there was just no way to access it once it was shadowed in the inner block.

I mentioned in the last chapter that there are situations where I avoid using `:=` because it can make it unclear what variables are being used. That's because it is very easy to accidentally shadow a variable when using `:=`. Remember, we can use `:=` to create and assign to multiple variables at once. Also, not all of the variables on the lefthand side have to be new for `:=` to be legal. You can use `:=` as long as there is at least one new variable on the lefthand side. Let's look at another program (see Example 4-2), which can be found on The Go Playground.

Example 4-2. Shadowing with multiple assignment

```
func main() {
    x := 10
    if x > 5 {
        x, y := 5, 20
        fmt.Println(x, y)
    }
    fmt.Println(x)
}
```

Running this code gives you:

```
5 20
10
```

Although there was an existing definition of `x` in an outer block, `x` was still shadowed within the `if` statement. That's because `:=` only reuses variables that are declared in the current block. When using `:=`, make sure that you don't have any variables from an outer scope on the lefthand side, unless you intend to shadow them.

You also need to be careful to ensure that you don't shadow a package import. We'll talk more about importing packages in Chapter 9, but we've been importing the `fmt` package to print out results of our programs. Let's see what happens when we declare a variable called `fmt` within our `main` function, as shown in Example 4-3. You can try to run it on The Go Playground.

Example 4-3. Shadowing package names

```
func main() {
    x := 10
    fmt.Println(x)
    fmt := "oops"
    fmt.Println(fmt)
}
```

When we try to run this code, we get an error:

```
fmt.Println undefined (type string has no field or method Println)
```

Notice that the problem isn't that we named our variable `fmt`, it's that we tried to access something that the local variable `fmt` didn't have. Once the local variable `fmt` is declared, it shadows the package named `fmt` in the file block, making it impossible to use the `fmt` package for the rest of the `main` function.

Detecting Shadowed Variables

Given the subtle bugs that shadowing can introduce, it's a good idea to make sure that you don't have any shadowed variables in your programs. Neither `go vet` nor `golangci-lint` include a tool to detect shadowing, but you can add shadowing detection to your build process by installing the `shadow` linter on your machine:

```
$ go install golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow@latest
```

If you are building with a Makefile, consider including `shadow` in the `vet` task:

```
vet:
        go vet ./...
        shadow ./...
.PHONY:vet
```

When you run `make vet` against the previous code, you'll see that the shadowed variable is now detected:

```
declaration of "x" shadows declaration at line 6
```

The Universe Block

There's actually one more block that is a little weird: the universe block. Remember, Go is a small language with only 25 keywords. What's interesting is that the built-in types (like `int` and `string`), constants (like `true` and `false`), and functions (like `make` or `close`) aren't included in that list. Neither is `nil`. So, where are they?

Rather than make them keywords, Go considers these *predeclared identifiers* and defines them in the universe block, which is the block that contains all other blocks.

Because these names are declared in the universe block, it means that they can be shadowed in other scopes. You can see this happen by running the code in Example 4-4 on The Go Playground.

Example 4-4. Shadowing `true`

```
fmt.Println(true)
true := 10
fmt.Println(true)
```

When you run it, you'll see:

```
true
10
```

You must be very careful to never redefine any of the identifiers in the universe block. If you accidentally do so, you will get some very strange behavior. If you are lucky, you'll get compilation failures. If you are not, you'll have a harder time tracking down the source of your problems.

You might think that something this potentially destructive would be caught by linting tools. Amazingly, it isn't. Not even `shadow` detects shadowing of universe block identifiers.

if

The `if` statement in Go is much like the `if` statement in most programming languages. Because it is such a familiar construct, I've used it in early sample code without worrying that it'd be confusing. Example 4-5 shows a more complete sample.

Example 4-5. `if` and `else`

```
n := rand.Intn(10)
if n == 0 {
    fmt.Println("That's too low")
} else if n > 5 {
    fmt.Println("That's too big:", n)
} else {
    fmt.Println("That's a good number:", n)
}
```

If you run this code, you'll find that it always assigns 1 to `n`. This happens because the default random number seed in `math/rand` is hard-coded. In "Overriding a Package's Name" on page 183, we'll look at a way to ensure a good seed for random number generation while demonstrating how to handle package name collisions.

The most visible difference between `if` statements in Go and other languages is that you don't put parenthesis around the condition. But there's another feature that Go adds to `if` statements that helps you better manage your variables.

As we discussed in the section on shadowing variables, any variable declared within the braces of an `if` or `else` statement exists only within that block. This isn't that uncommon; it is true in most languages. What Go adds is the ability to declare variables that are scoped to the condition and to both the `if` and `else` blocks. Let's take a look by rewriting our previous example to use this scope, as shown in Example 4-6.

Example 4-6. Scoping a variable to an `if` statement

```
if n := rand.Intn(10); n == 0 {
    fmt.Println("That's too low")
} else if n > 5 {
    fmt.Println("That's too big:", n)
} else {
    fmt.Println("That's a good number:", n)
}
```

Having this special scope is very handy. It lets you create variables that are available only where they are needed. Once the series of `if/else` statements ends, `n` is undefined. You can test this by trying to run the code in Example 4-7 on The Go Playground.

Example 4-7. Out of scope…

```
if n := rand.Intn(10); n == 0 {
    fmt.Println("That's too low")
} else if n > 5 {
    fmt.Println("That's too big:", n)
} else {
    fmt.Println("That's a good number:", n)
}
fmt.Println(n)
```

Attempting to run this code produces a compilation error:

```
undefined: n
```

Technically, you can put any *simple statement* before the comparison in an `if` statement. This includes things like a function call that doesn't return a value or assigning a new value to an existing variable. But don't do this. Only use this feature to define new variables that are scoped to the `if/else` statements; anything else would be confusing.

Also be aware that just like any other block, a variable declared as part of an `if` statement will shadow variables with the same name that are declared in containing blocks.

for, Four Ways

As in other languages in the C family, Go uses a `for` statement to loop. What makes Go different from other languages is that `for` is the *only* looping keyword in the language. Go accomplishes this by using the `for` keyword in four different formats:

- A complete, C-style `for`
- A condition-only `for`
- An infinite `for`
- `for-range`

The Complete for Statement

The first one we'll look at is the complete `for` declaration you might be familiar with from C, Java, or JavaScript, as shown in Example 4-8.

Example 4-8. A complete `for` statement

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

You would probably be unsurprised to find that this program prints out the numbers from 0 to 9, inclusive.

Just like the `if` statement, there are no parenthesis around the parts of the `for` statement. Otherwise, it should look very familiar. There are three parts, separated by semicolons. The first is an initialization that sets one or more variables before the loop begins. There are two important details to remember about the initialization section. First, you *must* use `:=` to initialize the variables; `var` is *not* legal here. Second, just like variable declarations in `if` statements, you can shadow a variable here.

The second part is the comparison. This must be an expression that evaluates to a `bool`. It is checked immediately *before* the loop body runs, after the initialization, and after the loop reaches the end. If the expression evaluates to `true`, the loop body is executed.

The last part of a standard `for` statement is the increment. You usually see something like i++ here, but any assignment is valid. It runs immediately after each iteration of the loop, before the condition is evaluated.

The Condition-Only for Statement

Go allows you to leave off both the initialization and the increment in a `for` statement. That leaves a `for` statement that functions like the `while` statement found in C, Java, JavaScript, Python, Ruby, and many other languages. It looks like Example 4-9.

Example 4-9. A condition-only `for` statement

```
i := 1
for i < 100 {
        fmt.Println(i)
        i = i * 2
}
```

The Infinite for Statement

The third `for` statement format does away with the condition, too. Go has a version of a `for` loop that loops forever. If you learned to program in the 1980s, your first program was probably an infinite loop in BASIC that printed HELLO to the screen forever:

```
10 PRINT "HELLO"
20 GOTO 10
```

Example 4-10 shows the Go version of this program. You can run it locally or try it out on The Go Playground.

Example 4-10. Infinite looping nostalgia

```
package main

import "fmt"

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

Running this program gives you the same output that filled the screens of millions of Commodore 64s and Apple][s:

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
...
```

Press Ctrl-C when you are tired of walking down memory lane.

If you run this on The Go Playground, you'll find that it will stop execution after a few seconds. As a shared resource, the playground doesn't allow any one program to run for too long.

break and continue

How do you get out of an infinite `for` loop without using the keyboard or turning off your computer? That's the job of the `break` statement. It exits the loop immediately, just like the `break` statement in other languages. Of course, you can use `break` with any `for` statement, not just the infinite `for` statement.

There is no Go equivalent of the `do` keyword in Java, C, and JavaScript. If you want to iterate at least once, the cleanest way is to use an infinite `for` loop that ends with an `if` statement. If you have some Java code, for example, that uses a `do/while` loop:

```
do {
    // things to do in the loop
} while (CONDITION);
```

The Go version looks like this:

```
for {
    // things to do in the loop
    if !CONDITION {
        break
    }
}
```

Note that the condition has a leading `!` to *negate* the condition from the Java code. The Go code is specifying how to *exit* the loop, while the Java code specifies how to stay in it.

Go also includes the `continue` keyword, which skips over the rest of the body of a `for` loop and proceeds directly to the next iteration. Technically, you don't need a `continue` statement. You could write code like Example 4-11.

Example 4-11. Confusing code

```
for i := 1; i <= 100; i++ {
    if i%3 == 0 {
        if i%5 == 0 {
            fmt.Println("FizzBuzz")
        } else {
            fmt.Println("Fizz")
        }
    } else if i%5 == 0 {
        fmt.Println("Buzz")
    } else {
        fmt.Println(i)
    }
}
```

But this is not idiomatic. Go encourages short `if` statement bodies, as left-aligned as possible. Nested code is difficult to follow. Using a `continue` statement makes it easier to understand what's going on. Example 4-12 shows the code from the previous example, rewritten to use `continue` instead.

Example 4-12. Using `continue` to make code clearer

```
for i := 1; i <= 100; i++ {
    if i%3 == 0 && i%5 == 0 {
        fmt.Println("FizzBuzz")
        continue
    }
    if i%3 == 0 {
        fmt.Println("Fizz")
        continue
    }
    if i%5 == 0 {
        fmt.Println("Buzz")
        continue
    }
    fmt.Println(i)
}
```

As you can see, replacing chains of `if/else` statements with a series of `if` statements that use `continue` makes the conditions line up. This improves the layout of your conditions, which means your code is easier to read and understand.

The for-range Statement

The fourth `for` statement format is for iterating over elements in some of Go's built-in types. It is called a `for-range` loop and resembles the iterators found in other languages. In this section, we will look at how to use a `for-range` loop with strings, arrays, slices, and maps. When we cover channels in Chapter 10, we will talk about how to use them with `for-range` loops.

You can only use a `for-range` loop to iterate over the built-in compound types and user-defined types that are based on them.

First, let's take a look at using a `for-range` loop with a slice. You can try out the code in Example 4-13 on The Go Playground.

Example 4-13. The `for-range` loop

```
evenVals := []int{2, 4, 6, 8, 10, 12}
for i, v := range evenVals {
    fmt.Println(i, v)
}
```

Running this code produces the following output:

```
0 2
1 4
2 6
3 8
4 10
5 12
```

What makes a `for-range` loop interesting is that you get two loop variables. The first variable is the position in the data structure being iterated, while the second is the value at that position. The idiomatic names for the two loop variables depend on what is being looped over. When looping over an array, slice, or string, an `i` for *index* is commonly used. When iterating through a map, `k` (for *key*) is used instead.

The second variable is frequently called `v` for *value*, but is sometimes given a name based on the type of the values being iterated. Of course, you can give the variables any names that you like. If there are only a few statements in the body of the loop, single letter variable names work well. For longer (or nested) loops, you'll want to use more descriptive names.

What if you don't need to use the key within your `for-range` loop? Remember, Go requires you to access all declared variables, and this rule applies to the ones declared as part of a `for` loop, too. If you don't need to access the key, use an underscore (_) as the variable's name. This tells Go to ignore the value. Let's rewrite our slice ranging code to not print out the position. We can run the code in Example 4-14 on The Go Playground.

Example 4-14. Ignoring the key in a `for-range` loop

```
evenVals := []int{2, 4, 6, 8, 10, 12}
for _, v := range evenVals {
    fmt.Println(v)
}
```

Running this code produces the following output:

```
2
4
6
8
10
12
```

Any time you are in a situation where there's a value returned, but you want to ignore it, use an underscore to hide the value. You'll see the underscore pattern again when we talk about functions in Chapter 5 and packages in Chapter 9.

What if you want the key, but don't want the value? In this situation, Go allows you to just leave off the second variable. This is valid Go code:

```
uniqueNames := map[string]bool{"Fred": true, "Raul": true, "Wilma": true}
for k := range uniqueNames {
    fmt.Println(k)
}
```

The most common reason for iterating over the key is when a map is being used as a set. In those situations, the value is unimportant. However, you can also leave off the value when iterating over arrays or slices. This is rare, as the usual reason for iterating over a linear data structure is to access the data. If you find yourself using this format for an array or slice, there's an excellent chance that you have chosen the wrong data structure and should consider refactoring.

When we look at channels in Chapter 10, we'll see a situation where a `for-range` loop only returns a single value each time the loop iterates.

Iterating over maps

There's something interesting about how a `for-range` loop iterates over a map. You can run the code in Example 4-15 on The Go Playground.

Example 4-15. Map iteration order varies

```
m := map[string]int{
    "a": 1,
    "c": 3,
    "b": 2,
}

for i := 0; i < 3; i++ {
    fmt.Println("Loop", i)
    for k, v := range m {
        fmt.Println(k, v)
    }
}
```

When you build and run this program, the output varies. Here is one possibility:

```
Loop 0
c 3
b 2
a 1
Loop 1
a 1
c 3
```

```
b 2
Loop 2
b 2
a 1
c 3
```

The order of the keys and values varies; some runs may be identical. This is actually a security feature. In earlier Go versions, the iteration order for keys in a map was usually (but not always) the same if you inserted the same items into a map. This caused two problems:

- People would write code that assumed that the order was fixed, and this would break at weird times.
- If maps always hash items to the exact same values, and you know that a server is storing some user data in a map, you can actually slow down a server with an attack called *Hash DoS* by sending it specially crafted data where all of the keys hash to the same bucket.

To prevent both of these problems, the Go team made two changes to the map implementation. First, they modified the hash algorithm for maps to include a random number that's generated every time a map variable is created. Next, they made the order of a `for-range` iteration over a map vary a bit each time the map is looped over. These two changes make it far harder to implement a Hash DoS attack.

There is one exception to this rule. To make it easier to debug and log maps, the formatting functions (like `fmt.Println`) always output maps with their keys in ascending sorted order.

Iterating over strings

As I mentioned earlier, you can also use a string with a `for-range` loop. Let's take a look. You can run the code in Example 4-16 on your computer or on The Go Playground.

Example 4-16. Iterating over strings

```
samples := []string{"hello", "apple_π!"}
for _, sample := range samples {
    for i, r := range sample {
        fmt.Println(i, r, string(r))
    }
    fmt.Println()
}
```

The output when we iterate over the word "hello" has no surprises:

```
0 104 h
1 101 e
2 108 l
3 108 l
4 111 o
```

In the first column, we have the index; in the second, the numeric value of the letter; and in the third, we have the numeric value of the letter type converted to a string.

Looking at the result for "apple_π!" is more interesting:

```
0 97 a
1 112 p
2 112 p
3 108 l
4 101 e
5 95 _
6 960 π
8 33 !
```

There are two things to notice. First, notice that the first column skips the number 7. Second, the value at position 6 is 960. That's far larger than what can fit in a byte. But in Chapter 3, we said that strings were made out of bytes. What's going on?

What we are seeing is special behavior from iterating over a string with a `for-range` loop. It iterates over the *runes*, not the *bytes*. Whenever a `for-range` loop encounters a multibyte rune in a string, it converts the UTF-8 representation into a single 32-bit number and assigns it to the value. The offset is incremented by the number of bytes in the rune. If the `for-range` loop encounters a byte that doesn't represent a valid UTF-8 value, the Unicode replacement character (hex value 0xfffd) is returned instead.

Use a `for-range` loop to access the runes in a string in order. The key is the number of bytes from the beginning of the string, but the type of the value is rune.

The for-range value is a copy

You should be aware that each time the `for-range` loop iterates over your compound type, it *copies* the value from the compound type to the value variable. *Modifying the value variable will not modify the value in the compound type*. Example 4-17 shows a quick program to demonstrate this. You can try it out on The Go Playground.

Example 4-17. Modifying the value doesn't modify the source

```
evenVals := []int{2, 4, 6, 8, 10, 12}
for _, v := range evenVals {
    v *= 2
}
fmt.Println(evenVals)
```

Running this code gives the following output:

```
[2 4 6 8 10 12]
```

The implications of this behavior are subtle. When we talk about goroutines in Chapter 10, you'll see that if you launch goroutines in a `for-range` loop, you need to be very careful in how you pass the index and value to the goroutines, or you'll get surprisingly wrong results.

Just like the other three forms of the `for` statement, you can use `break` and `continue` with a `for-range` loop.

Labeling Your for Statements

By default, the `break` and `continue` keywords apply to the `for` loop that directly contains them. What if you have nested `for` loops and you want to exit or skip over an iterator of an outer loop? Let's look at an example. We're going to modify our earlier string iterating program to stop iterating through a string as soon as it hits a letter "l." You can run the code in Example 4-18 on The Go Playground.

Example 4-18. Labels

```
func main() {
    samples := []string{"hello", "apple_π!"}
outer:
    for _, sample := range samples {
        for i, r := range sample {
            fmt.Println(i, r, string(r))
            if r == 'l' {
                continue outer
            }
        }
        fmt.Println()
    }
}
```

Notice that the label `outer` is indented by `go fmt` to the same level as the surrounding function. Labels are always indented to the same level as the braces for the block. This makes them easier to notice. Running our program gives the following output:

```
0 104 h
1 101 e
2 108 l
0 97 a
1 112 p
2 112 p
3 108 l
```

Nested `for` loops with labels are rare. They are most commonly used to implement algorithms similar to the pseudocode below:

```
outer:
    for _, outerVal := range outerValues {
        for _, innerVal := range outerVal {
            // process innerVal
            if invalidSituation(innerVal) {
                continue outer
            }
        }
        // here we have code that runs only when all of the
        // innerVal values were sucessfully processed
    }
```

Choosing the Right for Statement

Now that we've covered all of the forms of the `for` statement, you might be wondering when to use which format. Most of the time, you're going to use the `for-range` format. A `for-range` loop is the best way to walk through a string, since it properly gives you back runes instead of bytes. We have also seen that a `for-range` loop works well for iterating through slices and maps, and we'll see in Chapter 10 that channels work naturally with `for-range` as well.

Favor a `for-range` loop when iterating over all the contents of an instance of one of the built-in compound types. It avoids a great deal of boilerplate code that's required when you use an array, slice, or map with one of the other `for` loop styles.

When should you use the complete `for` loop? The best place for it is when you aren't iterating from the first element to the last element in a compound type. While you could use some combination of `if`, `continue`, and `break` within a `for-range` loop, a standard `for` loop is a clearer way to indicate the start and end of your iteration. Compare these two code snippets, both of which iterate over the second through the second-to-last elements in an array. First the `for-range` loop:

```
evenVals := []int{2, 4, 6, 8, 10}
for i, v := range evenVals {
    if i == 0 {
        continue
```

```
        }
        if i == len(evenVals)-2 {
            break
        }
        fmt.Println(i, v)
    }
```

And here's the same code, with a standard `for` loop:

```
    evenVals := []int{2, 4, 6, 8, 10}
    for i := 1; i < len(evenVals)-1; i++ {
        fmt.Println(i, evenVals[i])
    }
```

The standard `for` loop code is both shorter and easier to understand.

This pattern does not work for skipping over the beginning of a string. Remember, a standard `for` loop doesn't properly handle multibyte characters. If you want to skip over some of the runes in a string, you need to use a `for-range` loop so that it will properly process runes for you.

The remaining two `for` statement formats are used less frequently. The condition-only `for` loop is, like the `while` loop it replaces, useful when you are looping based on a calculated value.

The infinite `for` loop is useful in some situations. There should always be a `break` somewhere within the body of the `for` loop since it's rare that you want to loop forever. Real-world programs should bound iteration and fail gracefully when operations cannot be completed. As shown previously, an infinite `for` loop can be combined with an `if` statement to simulate the `do` statement that's present in other languages. An infinite `for` loop is also used to implement some versions of the *iterator* pattern, which we will look at when we review the standard library in "io and Friends" on page 233.

switch

Like many C-derived languages, Go has a `switch` statement. Most developers in those languages avoid `switch` statements because of their limitations on values that can be switched on and the default fall-through behavior. But Go is different. It makes `switch` statements useful.

For those readers who are more familiar with Go, we're going to cover *expression switch* statements in this chapter. We'll discuss *type switch* statements when we talk about interfaces in Chapter 7.

At first glance, `switch` statements in Go don't look all that different from how they appear in C/C++, Java, or JavaScript, but there are a few surprises. Let's take a look at a sample `switch` statement. You can run the code in Example 4-19 on The Go Playground.

Example 4-19. The `switch` statement

```
words := []string{"a", "cow", "smile", "gopher",
    "octopus", "anthropologist"}
for _, word := range words {
    switch size := len(word); size {
    case 1, 2, 3, 4:
        fmt.Println(word, "is a short word!")
    case 5:
        wordLen := len(word)
        fmt.Println(word, "is exactly the right length:", wordLen)
    case 6, 7, 8, 9:
    default:
        fmt.Println(word, "is a long word!")
    }
}
```

When we run this code, we get the following output:

```
a is a short word!
cow is a short word!
smile is exactly the right length: 5
anthropologist is a long word!
```

Let's go over the features of the `switch` statement to explain the output. As is the case with `if` statements, you don't put parenthesis around the value being compared in a `switch`. Also like an `if` statement, you can declare a variable that's scoped to all of the branches of the `switch` statement. In our case, we are scoping the variable `word` to all of the cases in the `switch` statement.

All of the `case` clauses (and the optional `default` clause) are contained inside a set of braces. But you should note that you don't put braces around the contents of the `case` clauses. You can have multiple lines inside a `case` (or `default`) clause and they are all considered to be part of the same block.

Inside `case 5:`, we declare `wordLen`, a new variable. Since this is a new block, you can declare new variables within it. Just like any other block, any variables declared within a `case` clause's block are only visible within that block.

If you are used to putting a `break` statement at the end of every `case` in your `switch` statements, you'll be happy to notice that they are gone. By default, cases in `switch` statements in Go don't fall through. This is more in line with the behavior in Ruby or (if you are an old-school programmer) Pascal.

This prompts the question: if cases don't fall through, what do you do if there are multiple values that should trigger the exact same logic? In Go, you separate multiple matches with commas, like we do when matching 1, 2, 3, and 4 or 6, 7, 8, and 9. That's why we get the same output for both `a` and `cow`.

Which leads to the next question: if you don't have fall-through, and you have an empty case (like we do in our sample program when the length of our argument is 6, 7, 8, or 9 characters), what happens? In Go, *an empty case means nothing happens*. That's why we don't see any output from our program when we use *octopus* or *gopher* as the parameter.

For the sake of completeness, Go does include a `fallthrough` keyword, which lets one case continue on to the next one. Please think twice before implementing an algorithm that uses it. If you find yourself needing to use `fallthrough`, try to restructure your logic to remove the dependencies between cases.

In our sample program we are switching on the value of an integer, but that's not all you can do. You can switch on any type that can be compared with `==`, which includes all of the built-in types except slices, maps, channels, functions, and structs that contain fields of these types.

Even though you don't need to put a `break` statement at the end of each `case` clause, you can use them in situations where you want to exit early from a `case`. However, the need for a `break` statement might indicate that you are doing something too complicated. Consider refactoring your code to remove it.

There is one more place where you might find yourself using a `break` statement in a `case` in a `switch` statement. If you have a `switch` statement inside a `for` loop, and you want to break out of the `for` loop, put a label on the `for` statement and put the name of the label on the `break`. If you don't use a label, Go assumes that you want to break out of the `case`. Let's look at a quick example. You can run the code in Example 4-20 on The Go Playground.

Example 4-20. The case of the missing label

```
func main() {
    for i := 0; i < 10; i++ {
        switch {
        case i%2 == 0:
            fmt.Println(i, "is even")
        case i%3 == 0:
            fmt.Println(i, "is divisible by 3 but not 2")
        case i%7 == 0:
            fmt.Println("exit the loop!")
```

```
            break
        default:
            fmt.Println(i, "is boring")
        }
    }
}
```

Running this code produces the following output:

```
0 is even
1 is boring
2 is even
3 is divisible by 3 but not 2
4 is even
5 is boring
6 is even
exit the loop!
8 is even
9 is divisible by 3 but not 2
```

That's not what we intended. The goal was to break out of the `for` loop when we got a 7. To do this, we need to introduce a label, just like we did when breaking out of a nested `for` loop. First, we label the `for` statement:

```
loop:
    for i := 0; i < 10; i++ {
```

Then we use the label on our break:

```
    break loop
```

You can see these changes on The Go Playground. When we run it again, we get the output that we expected:

```
0 is even
1 is boring
2 is even
3 is divisible by 3 but not 2
4 is even
5 is boring
6 is even
exit the loop!
```

Blank Switches

There's another, more powerful way to use `switch` statements. Just like Go allows you to leave out parts from a `for` statement's declaration, you can write a `switch` statement that doesn't specify the value that you're comparing against. This is called a *blank switch*. A regular `switch` only allows you to check a value for equality. A blank `switch` allows you to use any boolean comparison for each `case`. You can try out the code in Example 4-21 on The Go Playground.

Example 4-21. The blank `switch`

```
words := []string{"hi", "salutations", "hello"}
for _, word := range words {
    switch wordLen := len(word); {
    case wordLen < 5:
        fmt.Println(word, "is a short word!")
    case wordLen > 10:
        fmt.Println(word, "is a long word!")
    default:
        fmt.Println(word, "is exactly the right length.")
    }
}
```

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

```
hi is a short word!
salutations is a long word!
hello is exactly the right length.
```

Just like a regular `switch` statement, you can optionally include a short variable declaration as part of your blank `switch`. But unlike a regular `switch`, you can write logical tests for your cases. Blank switches are pretty cool, but don't overdo them. If you find that you have written a blank `switch` where all of your cases are equality comparisons against the same variable:

```
switch {
case a == 2:
    fmt.Println("a is 2")
case a == 3:
    fmt.Println("a is 3")
case a == 4:
    fmt.Println("a is 4")
default:
    fmt.Println("a is ", a)
}
```

you should replace it with an expression `switch` statement:

```
switch a {
case 2:
    fmt.Println("a is 2")
case 3:
    fmt.Println("a is 3")
case 4:
    fmt.Println("a is 4")
default:
    fmt.Println("a is ", a)
}
```

Choosing Between if and switch

As a matter of functionality, there isn't a lot of difference between a series of `if/else` statements and a blank `switch` statement. Both of them allow a series of comparisons. So, when should you use `switch` and when should you use a set of `if/else` statements? A `switch` statement, even a blank `switch`, indicates that there is some relationship between the values or comparisons in each case. To demonstrate the difference in clarity, let's rewrite our random number classifier from the section on `if` using a `switch` instead, as shown in Example 4-22.

Example 4-22. Rewriting `if/else` with a blank `switch`

```
switch n := rand.Intn(10); {
case n == 0:
    fmt.Println("That's too low")
case n > 5:
    fmt.Println("That's too big:", n)
default:
    fmt.Println("That's a good number:", n)
}
```

Most people would agree that this is more readable. The value being compared is listed on a line by itself, and all of the cases line up on the lefthand side. The regularity of the location of the comparisons makes them easy to follow and modify.

Of course, there is nothing in Go that prevents you from doing all sorts of unrelated comparisons on each case in a blank `switch`. However, this is not idiomatic. If you find yourself in a situation where you want to do this, use a series of `if/else` statements (or perhaps consider refactoring your code).

Favor blank `switch` statements over `if/else` chains when you have multiple related cases. Using a `switch` makes the comparisons more visible and reinforces that they are a related set of concerns.

goto—Yes, goto

There is a fourth control statement in Go, but chances are, you will never use it. Ever since Edgar Dijkstra wrote "Go To Statement Considered Harmful" in 1968, the `goto` statement has been the black sheep of the coding family. There are good reasons for this. Traditionally, `goto` was dangerous because it could jump to nearly anywhere in a program; you could jump into or out of a loop, skip over variable definitions, or into the middle of a set of statements in an `if` statement. This made it difficult to understand what a `goto`-using program did.

Most modern languages don't include `goto`. Yet Go has a `goto` statement. You should still do what you can to avoid using it, but it has some uses, and the limitations that Go places on it make it a better fit with structured programming.

In Go, a `goto` statement specifies a labeled line of code and execution jumps to it. However, you can't jump anywhere. Go forbids jumps that skip over variable declarations and jumps that go into an inner or parallel block.

The program in Example 4-23 shows two illegal `goto` statements. You can attempt to run it on The Go Playground.

Example 4-23. Go's `goto` has rules

```
func main() {
    a := 10
    goto skip
    b := 20
skip:
    c := 30
    fmt.Println(a, b, c)
    if c > a {
        goto inner
    }
    if a < b {
    inner:
        fmt.Println("a is less than b")
    }
}
```

Trying to run this program produces the following errors:

```
goto skip jumps over declaration of b at ./main.go:8:4
goto inner jumps into block starting at ./main.go:15:11
```

So what should you use `goto` for? Mostly, you shouldn't. Labeled `break` and `continue` statements allow you to jump out of deeply nested loops or skip iteration. The program in Example 4-24 has a legal `goto` and demonstrates one of the few valid use cases.

Example 4-24. A reason to use `goto`

```
func main() {
    a := rand.Intn(10)
    for a < 100 {
        if a%5 == 0 {
            goto done
        }
        a = a*2 + 1
    }
```

```
    fmt.Println("do something when the loop completes normally")
done:
    fmt.Println("do complicated stuff no matter why we left the loop")
    fmt.Println(a)
}
```

This example is contrived, but it shows how `goto` can make a program clearer. In our simple case, there is some logic that we don't want to run in the middle of the function, but we do want to run the end of the function. There are ways to do this without `goto`. We could set up a boolean flag or duplicate the complicated code after the `for` loop instead of having a `goto`, but there are drawbacks to both of these approaches. Littering your code with boolean flags to control the logic flow is arguably the same functionality as the `goto` statement, just more verbose. Duplicating complicated code is problematic because it makes your code harder to maintain. These situations are rare, but if you cannot find a way to restructure your logic, using a `goto` like this actually improves your code.

If you want to see a real-world example, you can take a look at the `floatBits` method in the file atof.go in the `strconv` package in the standard library. It's too long to include in its entirety, but the method ends with this code:

```
    overflow:
        // ±Inf
        mant = 0
        exp = 1<<flt.expbits - 1 + flt.bias
        overflow = true

    out:
        // Assemble bits.
        bits := mant & (uint64(1)<<flt.mantbits - 1)
        bits |= uint64((exp-flt.bias)&(1<<flt.expbits-1)) << flt.mantbits
        if d.neg {
            bits |= 1 << flt.mantbits << flt.expbits
        }
        return bits, overflow
```

Before these lines, there are several condition checks. Some require the code after the `overflow` label to run, while other conditions require skipping that code and going directly to `out`. Depending on the condition, there are `goto` statements that jump to `overflow` or `out`. You could probably come up with a way to avoid the `goto` statements, but they all make the code harder to understand.

You should try very hard to avoid using `goto`. But in the rare situations where it makes your code more readable, it is an option.

Wrapping Up

This chapter covered a lot of important topics for writing idiomatic Go. We've learned about blocks, shadowing, and control structures, and how to use them correctly. At this point, we're able to write simple Go programs that fit within the main function. It's time to move on to larger programs, using functions to organize our code.