GO
SUCCINCTLY

BY **MARK LEWIN**

Go Succinctly

By

By

Mark Lewin

Foreword by Daniel Jebaraj

Technical Reviewer: Matt Duffield

Copy Editor: Courtney Wright

Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.

Proofreader: John Elderkin

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The Succinctly series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Mark Lewin has been developing, teaching, and writing about software for more than 16 years. His main interest is web development in general and web mapping in particular. While working for ESRI, the world's largest GIS company, he acted as a consultant, trainer, and course author, and he was a frequent speaker at industry events. He currently works with a wide variety of open source mapping technologies and a handful of relevant JavaScript frameworks, including Node.js, Dojo, and jQuery.

Based in the UK, Mark currently teaches ArcGIS Server development for Geospatial Training Services and is the author of the *Google Maps API: Get Started* course for Pluralsight. By day, he writes MySQL developer courseware for Oracle.

Mark blogs about web mapping here and can be reached at mark@appswithmaps.net or on Twitter at @gisapps.

This is Mark's second book for Syncfusion. The first, *Leaflet.js Succinctly,* was published in 2016.

Chapter 1 Welcome

Who is this book for?

This book is for any developer familiar with a programming language such as Java or C# who is looking for a fast-paced but general introduction to Go.

While I cover a lot of ground in this short book, there are plenty of topics that I don't have the space to delve into. The Go language itself is quite small and concise, but there's a lot of depth to it, and the sheer volume of packages available in the Go Standard Library makes covering everything impossible.

My goal is to give you enough Go to inspire you to learn more. Take what I give you, enhance it, break it, work out how to fix it, and dive into the documentation and other resources available for truly immersing yourself in the language.

Go is gaining significant traction for very good reason: it is an excellent language for solving modern-day development problems. I predict that Go will become increasingly popular over the next few years, and any investment you put into learning it now will be rewarded in the future.

Code examples

All code examples in this book can be found on [Bitbucket](#).

Each sample can be identified by the code listing number given in this book. Furthermore, each code file has a link to the Go Playground sandbox environment so that you can run it on Google's servers without installing Go on your own machine (if that's what you want to do). The only exceptions are samples that require external resources because you cannot link to them from the Playground: the only communication that playground programs have with the outside world is via standard output.

We will talk more about the Go Playground in Chapter 2.

Acknowledgements

I would like to thank the team at Syncfusion for inviting me to write this book, as well as the editors and reviewers that made it look and read better than I could have managed on my own. Any book project, even one as small as this, involves a significant time investment, and for me that meant locking myself away in my office when I could have been spending time with my lovely wife Amanda and our six wonderful children: Ben, Marie, Samuel, Malachy, Patrick, and Grace. I'd like to extend my thanks to them for all their support and understanding. Ben, in particular, gets an extra vote of thanks for helping me with proofreading.

So with all that out of the way, let's talk about Go.

Chapter 2 Introducing Go

Welcome to Go

Whether you're an experienced programmer or a novice, you're going to love Go.

Go was conceived and born at Google in 2007, when three veteran engineers (Robert Griesemer, Rob Pike, and Ken Thompson) got together to build the language *they* wanted to use for systems programming. Google is a unique company, and the problems it faces are unique. With an untold number of computers, all networked together and running on different platforms and being developed in many different programming languages, no systems programming task is trivial at Google.

The rationale behind Go was to create a programming language that could be used to build robust and hugely scalable software, abstracting away some of the complexity of working in an environment using more traditional programming languages. Many people, myself included, really enjoy programming in Go, but the idea was that even if it you didn't fall in love with the language, you could at least appreciate the sensible design decisions that went into it. Creating Go was never about pushing completely new paradigms in programming language design; it was all about building a great language for the sort of challenges programmers face today.

So what challenges did these engineers identify, and how were those met by Go?

Slow builds

Even though computers are much faster these days, software builds for many programmers are still painfully slow. With Go, you can compile a substantial program in just a few seconds on a single computer. Everything compiles to a single binary file. There is no interpreter (like Python's or JavaScript's) and no reliance on a runtime environment like Java's JVM.

Dependency management

Managing dependencies is a huge issue for programmers today, and the old "header file" approach of languages such as C++ just doesn't scale very well. As programs have grown in size, working out when dependencies are no longer required has become very difficult.

Go's dependency management is baked into the language itself. If a package is listed as a dependency and no code within the program uses it, the Go compiler raises an error and no further compilation occurs until the dependency is removed. This guarantees that there are never any unwanted dependencies and keeps the code base lean. Go also builds dependent packages before the packages that depend on them, leading to much faster compilation because only a single file is opened by the compiler when an import occurs.

Static typing

"Hold on!" I hear you say. "Isn't there a big backlash against static typing in so-called 'modern' dynamically-typed languages like JavaScript and Python?"

Well yes, but dynamic typing comes with its own problems. With a statically typed language such as C++ or Java, you must specify the type of the data stored by a variable. The compiler checks variable assignments in your program and shouts at you if you're trying to store data of one type in a variable that only supports data of a different type. This can help weed out a lot of silly errors early on and avoid some painful debugging later. But advocates of dynamically typed languages applaud the flexibility and speed of development. Go offers the best of both worlds: it is statically typed, but features of the language make it appear to have some of the benefits of a dynamically typed language.

Concurrency

Concurrency means executing multiple routines simultaneously in order to speed up processing and offload effort. It's the way to make use of all those extra resources in the multicore CPUs that are ubiquitous nowadays. In many programming languages, concurrency is an afterthought and difficult to do well. Go provides great support for concurrent programming with Goroutines and channels.

Memory management

Unlike many compiled languages, you don't have to manage memory yourself. This has often been the cause of very hard-to-debug memory leaks that gradually bring systems to a halt. C and C++ are notorious for this: mismanaged memory leads to memory leaks, which are difficult to debug and will eventually bring your application to a standstill. Go manages memory for you. When resources are no longer required, Go's garbage collector gets rid of them and releases the memory back to the system.

Programmer consistency

Programmers will argue day and night about the best way to format code. With Go, there is no need to argue about the virtue of tabs versus spaces or different commenting styles because the Go guidelines make clear which way Go thinks you should do it. There's even a tool that will take your code and format it, Go-style, for you. All of the examples in this course have been run through the Go formatter. You can either call it explicitly, or, in most IDEs and editors, you can configure it to execute every time you save the file you are working on.

Tool chain

Go comes with many other out-of-the-box tools for documentation, testing, maintenance, and all the other time-consuming tasks that programmers must deal with.

Ease of learning

Google uses of C-style syntax, making it easy for anyone with experience of programming in other C-style languages such as Java, C#, and JavaScript to learn Go.

In short, Go is a modern programming language built to deal with modern programming challenges. It is small, easy to learn, and hugely powerful. However, it's not ideal for every type of project. Let's look at what Go is good at and where it is less useful.

What can I use Go for?

Go is good for:

- **Network applications, web servers, and distributed systems in general.** Go's support for concurrency is fantastic, and these applications rely on concurrency to scale well. Better still, Go's concurrency features are native to the language itself and not dependent on external libraries. Support for other requirements, such as HTTP, are available in Go's standard library.
- **Command-line applications and scripting.** Everything in Go compiles to a single binary file that can run lightning fast on just about any platform, with no external dependencies unless you expressly require them. What's more, Go programs can talk directly with C libraries and make system calls to the operating system. These features make Go the ideal choice for command-line applications and scripts.

Go is not so great at:

- **Desktop and graphical user interface applications.** Go's strength at running on multiple platforms is actually its downfall when it comes to writing desktop and GUI applications. While there are bindings for GTK, a library called `walk` for Windows applications, and work afoot at Google to build a cross-platform UI library called `gxui`, the Go community hasn't settled on an ideal approach for building GUI applications.
- **Very low-level programming.** Even though Go can make calls directly to operating system functions, it's not ideal for low-level work such as embedded systems and device drivers. A lot of this has to do with its heavy reliance on the operating system. However, there is work being done on a project called Ethos, which might make Go more viable for such applications in the future.

Now that we know a bit about the rationale behind Go and its strengths and weaknesses, let's talk about the language itself.

Chapter 3 Let's Go!

The Golang website

The Golang (short for Go Programming Language) website is your go-to resource (sorry, couldn't resist) for all things related to Go. From here, you can download the Go compiler and tools, access online documentation, and read all about what's new and wonderful in the world of Go.

Check it out now. Visit the Golang website and introduce yourself to the Go Gopher, who is the iconic mascot for the Go project. (And if you are really interested in the history behind the Go Gopher, read this post on the Go project blog.)
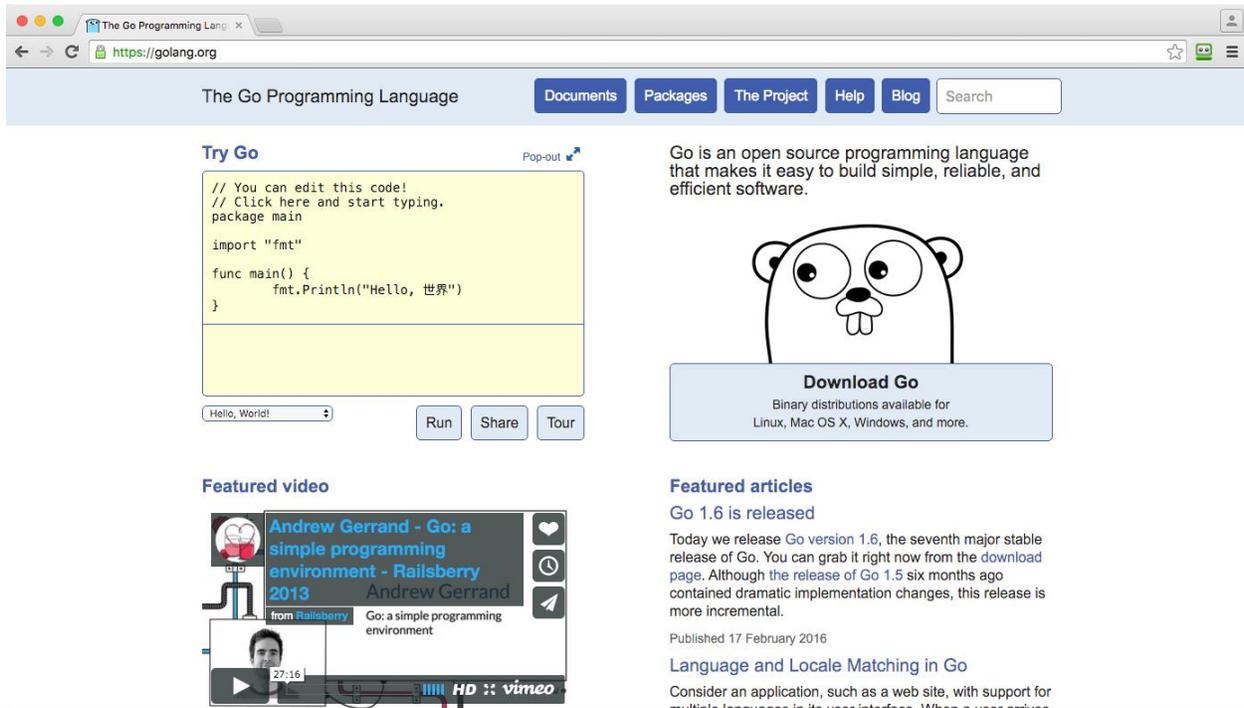
Figure 1: The Golang Homepage

Your first Go program

You don't even need to install Go to write your first Go program because you can try out Go directly from the Golang website. Let's do that now. After all, you might hate Go. I'd be very surprised and disappointed if you did, but I don't want to be the guy responsible for making you install lots of cruft on your machine that you'll never actually use. I'll leave it to the PC vendors to do that!

First, visit the Go Playground.

Next, select everything in the code window and delete it.

Then, type the following as the first line of your new Go program:

```
package main
```

That's it. No semicolons or other terminators required. But Go is a C-like language, and C and its derivatives use semicolons to tell the compiler where one line ends and another begins. Go uses semicolons too, but except for a few control structures, they don't usually appear in the source code. Instead, the lexer inserts them automatically as it scans your program looking for new lines and inferring the end of statements from the tokens it finds, which means you don't have to worry about terminating your statements. That's just one of the many things that Go does to make your life easier.

That **package** statement you just entered tells the Go compiler that this code file will live in the **main** package. Go organizes code into packages, and the most important package of all is **main**, because that's where your program will start. There are packages you write yourself, such as this one, and packages that you import from other sources.

Within the **main** package is a function called **main()**, which should come as no surprise if you have ever written any C or Java code. This is the entry point for your program.

So, press the return key a couple of times and enter:

```
func main() {

}
```

We're going to write a simple program that writes a "Hello, world!" message to the screen. To do that, we need to import a package from the Go standard library called **fmt**. In between your package declaration and your **main()** function, enter the following:

```
import (

        "fmt"

)
```

Now go back to your **main()** function and enter the following statement:

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

That line of code uses the **Printf()** function in the **fmt** package to display a line of text on the screen that is terminated by a new line character (**'\n'**).

Let's run it and see if it works. Just click the **Run** button at the top of the page and check the program output at the bottom. If you see your "Hello World!" message, your program is working fine. If not, compare your code to the screenshot in Figure 2.

Figure 2: The "Hello, world!" Program in the Go Playground

If everything worked as planned, your program was compiled on Google's servers and executed, displayed the "Hello, World!" message, and then exited. This process is extremely fast because Go's compiler is highly optimized. Yes, your program is only a few lines long and pretty trivial, but as you start to write more sophisticated code, the Go compiler will surprise you at just how fast it is—especially if you are a Java programmer.

I am reminded of an old joke at Java's expense here:

"*Knock! Knock!*"
"*Who's there?*"
"*   ...*"
"*   ...*"
"*   ...*"
"*   ...*"
"*   ...*"
"*... Java!*"

Now, I'd like you to break a couple of things just to get an idea of how Go works. First, delete the `fmt.Printf()` statement and run the program again. This time you should see an error, as shown in Figure 3. You'll see the culprit line number highlighted in pink and an error message at the bottom.

Figure 3: Unused Package Error

This is because Go, unlike many languages, keeps a strict eye on the packages you import and how they are used in your program. Go doesn't like you importing packages that you never use. It considers that a waste of resources and hates it so much that it raises an error. This keeps your programs nice and lean—not bloated by linked code that is never used.

Retype the line of code that prints the "Hello World!" message to your screen, but this time use a lowercase **p** for **printf()** and run the program again.

Figure 4: Unknown Function Error

The Go compiler throws another error. That's because Go is a case-sensitive language, and it won't forgive you if you get the capitalization wrong.

Fix up your program so that it's working correctly.

Another thing to know about Go is the way it deals with strings. In most languages, strings are just a bunch of bytes. In Go, they are "runes," which are really just integer values mapped to their Unicode counterparts. This makes it very easy to include foreign language characters in Go strings (and you can even use them in function names if you are so inclined).

For example, if we wanted to write "Hello, World!" in Telegu, we don't need to do anything special to our code:
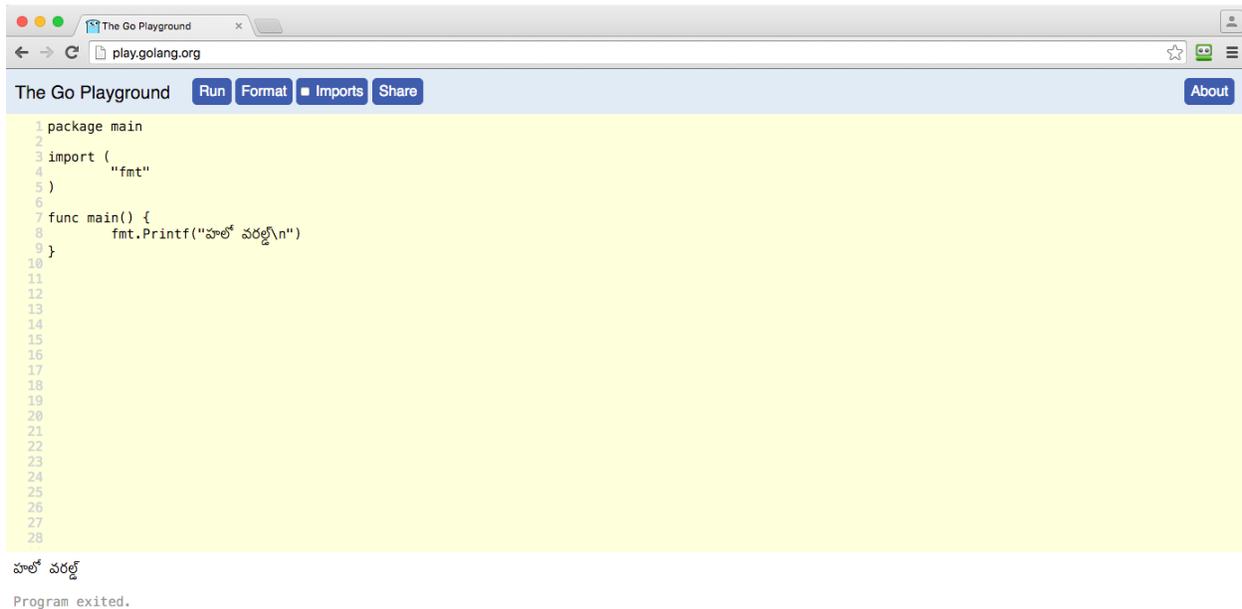
Figure 5: "Hello, World!" in Telegu

The Go Playground is a great place to experiment with the language, and you can even share the results of your labors with others (just click the **Share** button). However, as soon as you are ready to write something more substantial, you're going to need to download the tools you require to compile and execute code on your local machine.

Setting up your machine

There are a few things you need to set up on your machine before you're ready to start writing Go programs. The exact steps will depend on your operating system.

Text editor

You'll need an editor in which to write your code. Any editor will do, but if you can find one that supports syntax highlighting for Go, it will make life easier. You might have to do some Googling to find the appropriate syntax highlighting language packages for your chosen editor, but doing so will be worth it.

Atom or Sublime Text are very good cross-platform editors and have all sorts of add-ins you can use to compile and format your code. The old diehards emacs and vim are equally, if not more, capable. Ultimately, it's up to you which text editor you use because Go itself doesn't care.

If you're familiar with a certain IDE, you might be able to adapt it to use Go. For example, the IntelliJ IDEA platform has a plugin called go-ide.

I'm keeping things simple here and simply using a fairly simple text editor called TextWrangler on my Mac. It's not exciting, but it doesn't get in the way, either. Equivalent programs for Windows could be Notepad++, Notepad2, and PSPad, or, for Linux, Gedit, Geany, or Kate.

The terminal

The Go compiler and other tools are command-line applications, so you'll need access to a terminal program unless your fancy-schmancy editor can run the tools directly. This might be good when you're familiar with Go, but I'd suggest doing things via the command line when you're starting out because this gives you a better idea of how the tools work.

Windows

In Windows, you can open the terminal (or command prompt as it's known) by bringing up the Run dialog (hold down the **Windows logo key+R**) and then typing `cmd.exe`.

OS X

On a Mac, open **Finder** and navigate to **Applications** > **Utility** > **Terminal**. Pin it to the dock, because you'll be using it a lot in this book.

Linux

If you're running Linux you are doubtless familiar with the command line already and don't need any help from me.

The Go toolset

You can download the Go tools from the Golang website. Find the picture of the gopher (it's very hard to miss) and click on the **Download Go** button beneath it. Choose the version for your operating system from the "Featured Downloads" section. The current stable release of Go at the time of writing is 1.6, and all samples in this book are based upon this version. However, Google wants Go code to work across versions, so everything should still work if your version of Go is not the same as mine.

If you're on Windows or OS X, you can simply run the installer and you're good to go.

If you're on Linux, you can extract the **.tar.gz** file to /**usr/local/go** and no further setup is required. If you insist on installing it somewhere else, you'll need to set the **$GOROOT** environment variable accordingly. Consult the **README.md** file within the installer package for full instructions.

Test the installation by opening a terminal prompt and typing **go**. You should see a similar output to what is shown in Figure 6. When you run **go** without any other arguments, it simply lists the various tools and help available. You can see from the output that there are tools within the toolset for compiling, running, and formatting Go code; looking up documentation; and all the other tasks you will encounter as a Go programmer.
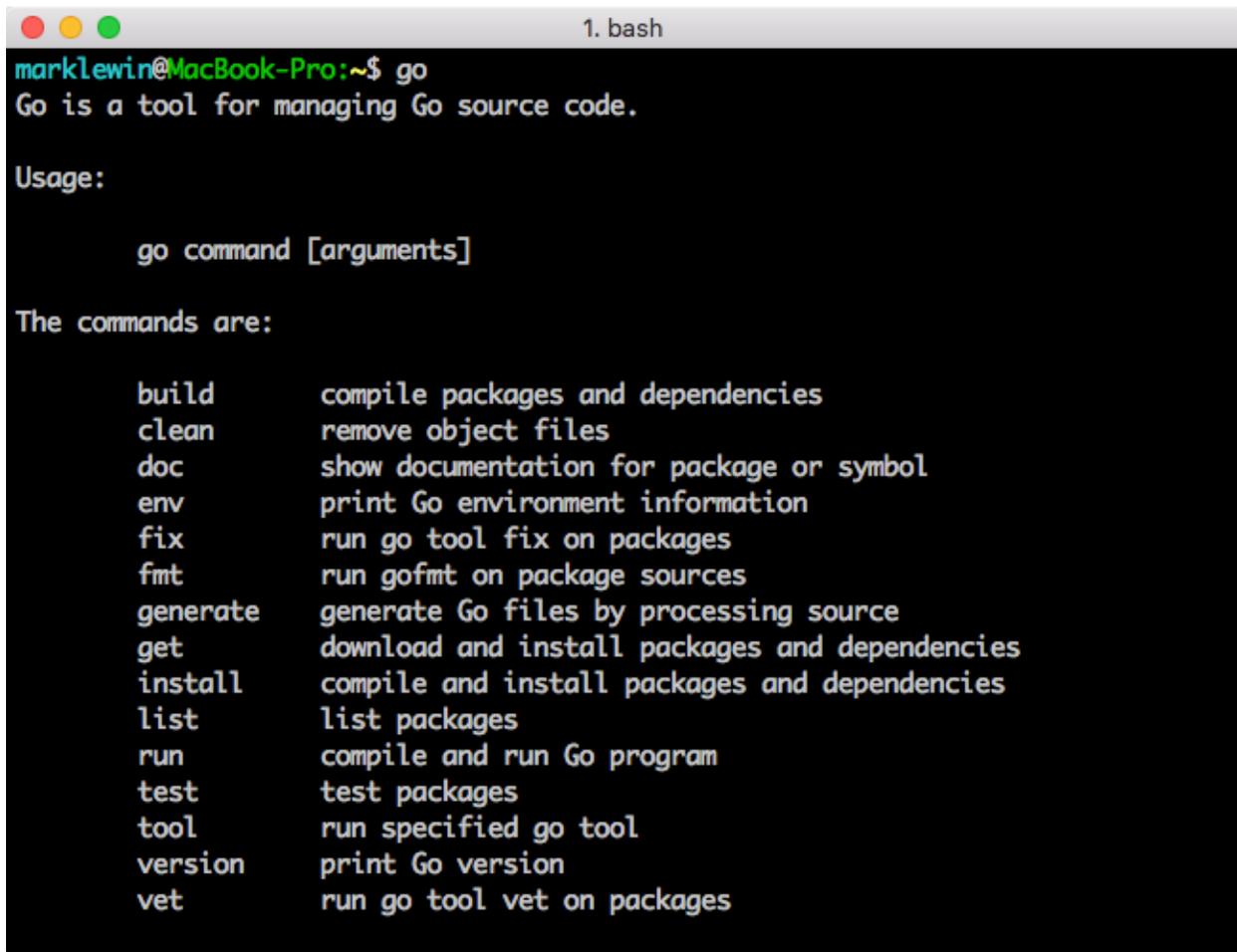
```
● ● ●                          1. bash

marklewin@MacBook-Pro:~$ go
Go is a tool for managing Go source code.

Usage:

        go command [arguments]

The commands are:

        build       compile packages and dependencies
        clean       remove object files
        doc         show documentation for package or symbol
        env         print Go environment information
        fix         run go tool fix on packages
        fmt         run gofmt on package sources
        generate    generate Go files by processing source
        get         download and install packages and dependencies
        install     compile and install packages and dependencies
        list        list packages
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
        vet         run go tool vet on packages
```

Figure 6: Checking the Installation of the Go Toolset

The Go workspace

In most programming environments, every code project has a separate workspace, each of which may be under source control. Go takes a different approach in that Go programmers usually keep all their Go code in a single workspace. Getting the structure of your workspace just right is important because the Go toolset expects things to be in certain places and gets very upset when you do things differently.

A Go workspace is simply a directory that contains three main subdirectories:

- `src`: Contains your Go source code files
- `pkg`: Contains packages
- `bin`: Contains executable programs

In practice, there will be further divisions within the `src` directory that map to different version control repositories, but for now we're going to keep things very simple.

Create a location on your machine for all your Go projects. This will be your workspace. For example, mine is in the `/Code/Go` folder within my home directory (`~/Code/Go`). The only requirement is that this must not be in the same path as your Go installation.

The Go tools need to know where to find your code, and they'll use an environment variable called `GOPATH` to do this. The `GOPATH` environment variable points to your Go workspace.

Use the following operating system-specific instructions to set the `GOPATH`. These are based on my environment, so you'll need to change `/Code/Go` to reflect the location of your own workspace.

Windows

Open the command prompt and type the following:

```
setx GOPATH %USERPROFILE%\Code\Go
```

This sets `GOPATH` to `C:\users\username\Code\Go` on more recent versions of Windows. Older versions might not support this command-line approach, but you can also set `GOPATH` from within the Control Panel (**System** > **Advanced** > **Environment Variables**).

OS X and Linux

You'll want to store the `GOPATH` setting so that your terminal remembers it. You usually do this in `~/.bash_profile`. Enter the following command at the terminal:

```
echo 'export GOPATH=$HOME/<path to my folder>…' >> ~/.bash_profile
```

Exit the terminal, restart it, and enter the following command to ensure that the `GOPATH` setting has been persisted:

```
env | grep GOPATH
```

This should display the current value of the `GOPATH` environment variable.

Migrating "Hello, World!"

Let's take the "Hello, World!" program we created in the Go Playground and get it working on our local machine.

First, within the workspace directory, create a new subdirectory called **src** and also one beneath it called **hello**. Bear in mind that the directory paths you need might differ from mine if your workspace is in a different location.

Windows

```
md src\hello
```

OS X and Linux

```
mkdir –p src/hello
```

Within **/src/hello**, create a file called **main.go**. Note that you can call this file whatever you like, but because it is the location of our **main()** function, **main.go** seems as a good a name as any.

Enter the "Hello, World!" code from the Go playground into the **main.go** file in your text editor of choice and save it.

Code Listing 1

```go
package main

import (
        "fmt"
)

func main() {
        fmt.Printf("Hello, world!\n")
}
```

In order to run the program, you will need to use the **go** tool. Open your terminal, change to the **src/hello** directory, and enter the following command:

```
go run main.go
```

This command compiles the **main.go** file and, if it is successful in doing so, launches it. If your program displays "Hello, world!" in your terminal that means you have succeeded. Give yourself a pat on the back. If it fails, look at the errors generated by the compiler and try to fix them in your code before executing **go run main.go** again. If you manage it, give yourself an extra pat on the back for your debugging skills.

Formatting Go source code

Unlike many languages, Go is very picky about how you format your code. This gives you less of a chance to show off your artistic nature, but it does prevent arguments during code reviews at work and makes your code (and everyone else's) easier to read and, therefore, to maintain.

Go code has rules (that you must obey) and conventions (that they want you to obey). This is very different from Java and other languages derived from C. One of the biggest shocks programmers from these languages discover when they come to Go is how Go enforces rules on the use of curly braces.

In C and Java, when you have a block of code as part of a function or control structure (**if**, **for**, etc.), these are contained by curly braces. The C or Java compilers don't care whether the initial brace appears. For example:

```
function myFunction() {

    if (loggingEnabled == true) {

        log.output("I'm in myFunction()");

    }
    …
}
```

This code can equally be written like this:

```
function myFunction()
{

    if (loggingEnabled == true)
    {

        log.output("I'm in myFunction()");

    }
    …
}
```

But not in Go. Go insists that the opening brace is always on the same line as the statement preceding it. Most C and Java programmers use this style anyway, but they are surprised when Go raises an error if they depart from it.

Go also complains about the omission of curly braces when the body of a control structure contains only one line of code. Go insists upon it. For example, the following approach is illegal in Go:

```
if zombieApocalypse == true
        fmt.Println("Run for the hills")
```

Instead, it should be written as:

```
if zombieApocalypse == true {
        fmt.Println("Run for the hills")
}
```

Where there are no rules there are conventions, such as:

- Go uses tabs rather than spaces to indent code.
- When you import multiple packages using an import statement, Go likes them to be in alphabetical order.
- When you create custom types, Go likes the field names and types to be lined up nicely.

Because these conventions are hard to remember at first, and because the creators of Go want to make it easy for you to adopt their programming style, there is a tool called **fmt** in the toolset you can use to ensure that your code is very "Go-like."

Execute the tool from the command line as follows:

```
go fmt /path/to/your/package
```

For example, if this is the source file (**hello.go**):

```
package              main


import     "fmt"


func main() {
        i := 5
            fmt.Println((i))
    fmt.Printf(             "hello, world\n"    )
}
```

Then, after running **go fmt hello.go**, the output will appear as follows:

```
package main


import "fmt"
```

```
func main() {
  i := 5
  fmt.Println((i))
  fmt.Printf("hello, world\n")
}
```

Much nicer!

It's good to get into this habit, especially when working on projects with other coders who will expect your code to be formatted the "Go way." You can include it in your build process, and some editors have plugins that can automatically format your code each time you save a source file.

Getting help with Go

No matter how great a coder you are, you can't be expected to remember every intricacy of a programming language, even one as concise as Go. You're going to need help, and the help documentation in Go is very useful.

There are two main ways to access help. First, you can visit the **Documents** link on the Golang website. It contains an introduction, how-to guides and articles, and comprehensive reference documentation. You can also catch up on the latest Go news with the Go blog and listen to a number of recorded talks and presentations about Go. Also check out the forums, mailing lists, and IRC channels available [here](#).

You can also look up information about Go commands directly in Go itself. For example, to get information about the entire **fmt** package, enter the following at a terminal prompt:

```
godoc fmt
```

To drill into information about a particular Go type, variable, constant, or a function such as **Printf()**, enter the following:

```
godoc fmt Printf
```

If the documentation doesn't provide the answers you need, that is because Go is an open source project, which means you can go directly to the source code itself. However, I think you'll be pleased with the quality of the Go documentation.

Documenting your own code

You can also use the **godoc** tool to document your own packages. While this is beyond the scope of this book, all it requires are comments within your code, formatted in special ways depending on the type of information you are trying to impart. The syntax for creating comments in Go is the same as other C-style languages:

```
// This is a line comment that will be ignored by the compiler

/* This is a block comment.
It can span several lines and
will also be ignored by the compiler. */
```

Get into the habit of adding comments to your code. It's useful for other programmers, and also for you when you revisit code you wrote a while ago and no longer understand it!

That concludes our crash course in writing, compiling, running, and getting help with Go code. In subsequent chapters, we'll dig into the features of the language itself.

Recommended Go tool: `Golint`

This is another tool that I find very useful. It's not part of the standard toolset, but can be installed from GitHub.

Golint is a linter for Go source code. It differs from **go fmt** in that **go fmt** reformats source code, whereas **Golint** generates a list of what it perceives to be style misdemeanors. I find I learn more about what Go expects from me stylistically when it tells me what's wrong instead of just reformatting my code for me.

Install **Golint** by running:

go get github.com/golang/lint/golint

For usage instructions, refer to the online documentation.

Chapter 4 Variables, Constants, and Assignments

Statically vs. dynamically typed languages

Go is a statically typed programming language, meaning that when you define a variable, Go expects you to tell it what type of data it will hold. You can't then suddenly decide to store a different type of data in the same variable.

Wars have been fought, marriages have failed, and friendships been sundered as a result of programmers arguing about which is best: statically typed languages like Go, or dynamically typed languages like JavaScript. (Okay, I'm exaggerating. But nonetheless, programmers are definitely divided into camps over this.)

Go attempts to bridge the gap by being a statically typed language that nonetheless can be used in ways that makes it "feel" more like a dynamic language.

Variable assignment

The best way to demonstrate this is with an example. Let's take the "Hello, world!" program we built in the previous chapter and amend it so the message it displays is stored in a variable rather than hard-coded.

Code Listing 2

```go
package main

import (
        "fmt"
)

func main() {
        var msg string = "Hello, world!\n"
        fmt.Printf(msg)
}
```

Note the way we declare variables in Go. We use the **var** keyword, followed by the name of the variable (**msg)**, followed by the type (in this instance, **string**), and then finally we assign the value ("Hello World"). This might look a bit back-to-front compared to most statically typed languages, but it's the way Go likes to do things.

When we run it, we get exactly the same output we had before.

If we are initializing the variable at the point of declaration, we can omit the type and Go will infer it from the value we have assigned to it. So we can use the form:

var myVar = "This will use the string data type"

Go also provides us with an even more concise way of declaring a variable and assigning a value to it. Consider the following code:

Code Listing 3

```go
package main

import (
        "fmt"
)

func main() {
        msg := "Hello, world!\n"
        fmt.Printf(msg)
}
```

In this example, we've done away with the type declaration *and* the **var** keyword. Using this syntax feels very much like working with a dynamically typed language. However, we are not. In the presence of the **:=** operator, Go knows that we're assigning a value to a variable, and it infers the data type from the assignment value, just as before. If we then try to reassign a different data type to **msg**, we'll get an error, proving that no matter how "dynamic" Go might look at times, it is still a statically typed language:

Code Listing 4

```go
package main

import (
        "fmt"
)

func main() {
        msg := "Hello, world!\n"
        msg := 9
        fmt.Printf(msg)
}
```

./main.go:9: no new variables on left side of :=

./main.go:9: cannot use 9 (type int) as type string in assignment

In order to use the numerical value of 9, we must assign it to a different variable of numeric type. Let's change our program to use two variables and output them both in the same **print** statement.

Code Listing 5

```go
package main

import (
        "fmt"
)

func main() {
        msg := "Today's prize-winning entry is %d\n"
        winner := 9
        fmt.Printf(msg, winner)
}
```

Today's prize-winning entry is 9

Here we're using string interpolation (also known as variable substitution or variable expansion) to replace a placeholder (**%d**) with the value of the variable **winner**.

Not all data is variable—sometimes we want to ensure that a value remains unchanged throughout a program. We can use the **const** keyword for that, and a **const** can appear anywhere a **var** can.

When you are declaring constants, you don't have to specify the data type. You can, but you don't have to. That's because constants are treated differently in Go. When you don't specify a data type for a constant, Go gives it a special "undefined" data type and converts it to an appropriate data type at the point it is used. This is powerful, because it means that constants can be accessed widely throughout your program without having to worry about mismatched data types.

Scope

If we declare constants or variables within a function body, they'll only be available within that function. We can make them available to all functions in the **main** package by declaring them at the top level.

But remember that if we declare constants or variables at the top level, we must use the **var** or **const** keywords—we cannot use the shortcut variable assignment operator **:=**. We can group several assignments together at the top level by surrounding them with parentheses, as shown in Code Listing 6.

Code Listing 6

```go
package main

import (
        "fmt"
)

const (
        prizeDay = "Wednesday"
        prizeFund = 10000
)

func main() {
        msg := "%s's prize-winning entry is %d and wins %d!!!\n"
        winner := 9
        fmt.Printf(msg, prizeDay, winner, prizeFund)
}
```

Wednesday's prize-winning entry is 9 and wins 10000!!!

📝 *Note: We use `%s` to interpolate a string and `%d` to interpolate an integer. See a full list of interpolation verbs here.*

We can also scope variables in the same way:

Code Listing 7

```go
package main

import (
        "fmt"
)
```

```go
var (
        prizeDay = "Wednesday"
        prizeFund = 10000
)

func main() {
        prizeDay = "Thursday"
        prizeFund = 50000
        msg := "%s's prize-winning entry is %d and wins %d!!!\n"
        winner := 9
        fmt.Printf(msg, prizeDay, winner, prizeFund)
}
```

Thursday's prize-winning entry is 9 and wins 50000!!!

Because **prizeDay** and **prizeFund** are variables, we can change their values. If we had attempted to do that when they were defined as constants, we would have received the following error:

./main.go:13: cannot assign to prizeDay

./main.go:14: cannot assign to prizeFund

./main.go:14: cannot use 50000 (type int) as type untyped number in assignment

Now that we've seen how types are declared, let's have a look at the range of data types available.

Chapter 5 Basic Data Types

Basic data types in Go consist of integers, floating-point numbers, strings, and Booleans. These are the Go "primitives" from which the more complex types are built.

Numeric data types

Integers

Integers are simply numbers without a decimal part. In simple terms, they are "whole numbers." So 5, 34, and 7,424 are integers. 3.14, 22.7, and 0.01098 are not.

Although we like to think of numbers in terms of a 10-based numbering system, computers store them in binary form. The size of the number a computer can store depends on the amount of memory space allocated to the task. For example, a 4-bit integer can store 0000 (zero), 0001 (one), 0011 (three), and so on up to 1111 (15). So a 4-bit integer can represent the numbers zero to 15.

Go's integer data types range in storage capacity from 8 bits to 64 bits. If you need that level of control over memory allocation, you can choose from **uint8**, **uint16**, **uint32**, **uint64**, and from **int8**, **int16**, **int32**, and **int64**.There are also variations called **uint**s, which store only unsigned integers: *bytes* (equivalent to **uint8**) and *runes* (equivalent to **uint32**). However, gone are the days where memory is limited for many applications, and for most use cases the simple **int** data type will be more than adequate. It corresponds to either **int32** or **int64** and can store numbers from -2147483648 to 2147483647 and from -9223372036854775808 to 9223372036854775807, respectively.

If you need to work with larger integers than that, consider using the **uint** data type, which is either **uint32** or **uint64** and can store 0 to 4294967295 or 0 to 18446744073709551615, respectively.

Floating-point numbers

Floating-point numbers, or "real numbers," contain a decimal part. The important thing to realize is that they are often inexact and their accuracy depends on the number of decimal places the computer can store. As with integers, there are several different data types you can choose from, all offering different levels of precision, although the storage mechanism is not the same as for integer values. For the vast majority of cases, stick with **float32** and **float64** and consult the documentation if you have special requirements.

Working with numeric data

To assign an integer value to a variable, simply specify the number. To force a variable to store a number as a floating-point value, ensure that you specify the decimal component, even if it is zero.

Code Listing 8

```go
package main

import (
        "fmt"
)

func main() {
        int1 := 2
        int2 := 3

        fp1 := 22.0
        fp2 := 7.0

        fmt.Printf("%d + %d = %d\n", int1, int2, int1 + int2)
        fmt.Printf("%f / %f = %f\n", fp1, fp2, fp1 / fp2)
}
```

```
2 + 3 = 5

22.000000 / 7.000000 = 3.142857
```

Note that even though we are sure that the values of **fp1** and **fp2** are exact, the result of **fp1/fp2** is inexact, and the accuracy depends on the data type being used to hold it. Because we have allowed Go to infer the type for us, what actual type is Go using?

We can discover this by using Go's **reflect** library, which allows Go programs to interrogate themselves for information.

Code Listing 9

```go
package main

import (
  "fmt"
  "reflect"
)

func main() {
        fp1 := 22.0
        fp2 := 7.0
```

```
    result := fp1/fp2
    fmt.Printf("%f / %f = %f\n", fp1, fp2, result)

  fmt.Printf("fp1 is type: %s\n", reflect.TypeOf(fp1))
  fmt.Printf("fp2 is type: %s\n", reflect.TypeOf(fp2))
  fmt.Printf("result is type: %s\n", reflect.TypeOf(result))
}
```

22.000000 / 7.000000 = 3.142857

fp1 is type: float64

fp2 is type: float64

result is type: float64

If you want to specify the type of your float explicitly (such as **float32**), you could use one of the following constructs:

var fp1 float32;

fp1 = 3.14

or

fp1 := float32(3.14)

If you want to cast from one type to another, use the second form shown previously:

i := 373

f := float64(i)

u := uint(f)

If you declare a numeric value but don't give it a value, it is automatically assigned the value of zero.

Booleans

A Boolean is a special type of integer that has only two values, usually denoted by **true** and **false**. Boolean values are used for logical operations.

When you define a Boolean variable without assigning a value to it, it is given the default value of **false**.

As well as checking the value held by a Boolean variable, you can also use Boolean values with the three logical operators **&&** (**and**), **||** (**or**), and **!** (**not**). The following example shows how these work:

```
var trueValue bool
trueValue = true
falseValue := false
fmt.Println(trueValue && trueValue)    // true
fmt.Println(trueValue && falseValue)   // false
fmt.Println(trueValue || trueValue)    // true
fmt.Println(trueValue || falseValue)   // true
fmt.Println(!trueValue)                // false
```

Strings

Strings are absolutely fundamental to computer programs in all languages, and Go provides a lot of support for creating and manipulating strings.

You can define a string in Go by surrounding it with double quotes, as we've seen many times already, or by backquotes (`` ` ``). The difference is that if you use double quotes, you cannot include newlines. However, double-quoted strings do allow you to insert special "escape characters," such as **\n** for a carriage return and **\t** for a tab. Anything inside backquotes is used completely literally. For example:

```
str1 := "There is a tab between this\tand this"
str2 := `There is a tab between this\tand this`
fmt.Println(str1)
fmt.Println(str2)
```

This produces the following:

```
There is a tab between this    and this
There is a tab between this\tand this
```

If you declare a string variable and don't assign a value, Go gives it a default value of an empty string (**""**).

Common string operations

Concatenation

You can join two or more strings together to create a new, longer string by using the concatenation operator **+**. Note that, as with most languages, Go strings are immutable. The sequence of bytes in a string never changes, but you can always assign a new value to a string variable.

```
str1 := "This is a string "
str2 := " which is now much longer"
fmt.Println(str1 + str2)
```

This produces:

```
This is a string which is now much longer
```

The backquotes approach is very useful when you want to include double quotes within the string:

```
str1 := `I can include "double quotes" in this string without any issues`
str2 := "The use of "double quotes" within this string will cause a compiler
error"
```

Length

You can get the length of a string by using the **len()** function.

```
str := "This is a string"
fmt.Println(len(str)) // Displays 17
```

Substrings

You can return a character or characters from specific positions in the string by including square brackets, which contain a starting position and ending position separated by a colon. The first character in a string is at position zero, so the total number of positions equals **(len(str)-1)**. If you want the starting position to be either the at beginning or at the end of the string, just omit the position number.

```
str := "The elephants of spring are tickling my frying spoon"
fmt.Println(str[0:4])   // "The "
fmt.Println(str[17:23]) // "spring"
fmt.Println(str[:13])   // "The elephants"
fmt.Println(str[31:])   // "frying spoon"
```

Comparing strings

You can compare strings with the usual comparison operators: `==` (for equality), `<` (less than), `>` (greater than), and so on. The comparison is byte by byte, and the result depends on the natural lexicographic ordering (namely, how the strings might appear in a dictionary). For this example, we can use Go's **if/else** construct, which is very similar to how you might use **if/else** in another language, but note that in Go, you don't need parentheses around the conditions.

```
str1 := "abc"
str2 := "abd"
```

```
if str1 > str2 {
        fmt.Println("abc is greater than abd")
} else {
        fmt.Println("abd is greater than abc")
}
```

This produces:

```
abd is greater than abc
```

We'll look more at the **if/else** statement and the other Go control structures in the next chapter.

Converting between numbers and strings

Because Go is a statically typed language, you cannot use strings where you would expect a number, and vice versa.

For example:

```
int1 := 48
str1 := "My number is "
str1 = str1 + int1
// "invalid operation: str1 + int1 (mismatched types string and int)"
str2 := "4"
result := 48 / str2
fmt.Println(result)
// "cannot convert 48 to type string"
// "invalid operation: 48 / str2 (mismatched types int and string)"
```

The solution is to use the functions in the `strconv` package, which will provide a number of different ways for converting between strings and other data types. The most common functions are:

- **strconv.Atoi()** : Converts a string to an integer
- **strconv.Itoa()** : Converts an integer to a string

The **itoA()** function always works, so there is no facility for returning an error. But the **Atoi()** function returns two values: the converted value, and an object containing information about any errors resulting from the conversion. So we need to assign two variables to receive those returned values:

```
i,err := strconv.Atoi("32")
```

If we're not interested in whatever information **err** holds (that is, if we're confident that the conversion is straightforward and not going to cause an error condition), then by defining it we are duty-bound by Go to use it somewhere in our program. If we don't, we'll get a compiler error:

```
err declared and not used
```

The solution is to use another Go idiom: the empty variable (_). This will provide the "placeholder" for the returned value, but will allow us to ignore it:

```
i, _ := strconv.Atoi("32")
```

💡 *Tip: Use the empty variable (_) anywhere Go requires you to declare a variable that you don't intend to use anywhere else in your program. Otherwise, you will get a compilation error (Go likes to keep a tidy house and hates anything that is declared and never used).*

The result of fixing these data-type issues can be seen in Code Listing 10:

Code Listing 10

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
  int1 := 48
  int1_as_string := strconv.Itoa(int1)
  str1 := "My number is "
  str1 = str1 + int1_as_string
  fmt.Println(str1)

    str2 := "4"
  str2_as_integer, _ := strconv.Atoi(str2)
```

```
  result := 48 / str2_as_integer
  fmt.Printf("%s / %s is %d\n", int1_as_string, str2, result)
}
```

My number is 48

48 / 4 is 12

There are many other functions for converting strings to and from other data types within the **strconv** package:

- **ParseBool()**, **ParseFloat()**, **ParseInt()**, and **ParseUint()** convert strings to values.
- **FormatBool()**, **FormatFloat()**, **FormatInt()**, and **FormatUnit()** convert values to strings.

Consult the **strconv** package documentation for a full list.

Chapter 6 Control Structures

At the end of the last chapter, we saw how we could use an **if/else** statement to affect the execution path of a program. This is known as *conditional processing*, in which the path the program takes depends on the result of an expression.

We'll look at **if/else** in a little more depth in this chapter (you'll be pleased to know that you have already seen most of what it has to offer), together with its cousin the **switch** statement.

Sometimes you'll want your program to repeat a certain task a set number of times. Like most languages, Go provides a number of different alternatives for doing this, and we'll consider these, too.

In doing both of these things, we'll look at some other features of Go, such as writing functions and accepting arguments from the command line.

Using if/else

The **if/else** statement allows us to do one of two different things, depending on a particular condition:

```
if a == 1 {
  // a equals 1, so do this
} else {
  // a is not equal to 1, so do this instead
}
```

We can add further **if/else** statements to this if we want to conduct further tests. For example:

```
if a == 1 {
  // do this: a is 1
} else if a == 2 {
  // do this: a is 2
} else if b == 1 {
  // do this: b is 1, a is not 1 or 2
} else {
  // do this: b is not 1 and a is not 1 or 2
}
```

Let's demonstrate this using an example, which will also give us a chance to cover a couple of features that Go supports, namely command-line arguments and functions.

Our program should allow us to input a set of four numbers, and it will tell us if each of those numbers is odd or even.

First, we need a way to input numbers. We can do this by passing the four numbers we are interested in to our Go program at runtime as arguments. For example:

```
go run main.go 12 4 7 93
```

To access those arguments, we need to use the **os** package and retrieve its **Args** property. **Args** is a special type of variable called a *slice* that we'll consider in the next chapter. For now, just think of it as a collection of items that will include in its first position the directory path of our **main.go** program, and then, in subsequent positions, each of the arguments that we passed to the program at the command line.

Because we're only interested in the arguments we passed, we can safely ignore the first element in **Args**. This is the element in position zero because, like strings, elements in **Args** are numbered starting at zero.

Code Listing 11

```go
package main

import (
        "fmt"
        "os"
)

func main() {

    val1 := os.Args[1]
    val2 := os.Args[2]
    val3 := os.Args[3]
    val4 := os.Args[4]

    fmt.Printf("%s %s %s %s\n", val1, val2, val3, val4)
}
```

This program allows us to call our program with the four numbers we are interested in as parameters like so:

```
go run main.go 1 45 67 87
```

It then displays the numbers that we passed to it (as strings):

```
1 45 67 87
```

Now we can be sure that the program is accepting the numbers we are passing into it and storing those in variables we can use to access them.

The next thing we need to do is test each of those numbers to see if they are odd or even. To do that, we need to divide them by two and check to see if there is a remainder. Go provides the modulus (**%**) operator for that, which divides the first operand by the second operand and returns only the remainder. For example:

```
4 % 2 = 0
3 % 2 = 1
13 % 3 = 1
```

But we have four numbers, and it would be annoying to code the same test on each of them. Let's suppose that at some stage we want to expand our program to accept a thousand numbers, or even a million (although we probably won't be relying on the command line for that).

We'll code the test just once and put it into a function, separate from but like our **main()** function, and we'll call that instead. For now, we'll just call the function once for each of our numbers, but later in this chapter, when we start looking at loops, we'll tidy that up to accept an unlimited number of numbers as arguments.

Functions are defined by the keyword **func**, require a name and optionally a set of parameter values enclosed by parentheses, and, if the function returns a value, the type of value it returns. For example:

```
func myFunction(myString string, myNum int) float32 {

    // some code that creates a float32 value called myFloat

    return myFloat

}
```

This function is called **myFunction**, accepts two parameters—a string called **myString** and an integer value called **MyNum**—and returns a value of **float32** data type.

If your function does not require parameters, include the parentheses but leave them empty (like we have with **main()**). We need our function to accept a single parameter, which is the number we want to test.

We also want the function to return either **odd** or **even** when it is called, depending on the test results. We return a value to the calling function using the **return** keyword followed by the value we want to return. We'll decide which value to return using an **if/else** statement. Here's how we'll code the function:

```
func oddOrEven(value string) string {

    num, _ strconv.Atoi(value)

    if num % 2 > 0 {

        return "even"
```

```
        } else {
                return "odd"

        }
}
```

Note how we've used the **strconv.Atoi()** function to convert the input parameter, which is of type **string** to an integer, so that we can use the modulus operator (%) on it.

Here's what we end up with:

Code Listing 12

```go
package main

import (
        "fmt"
        "os"
        "strconv"
)

func main() {

  fmt.Printf("%s is %s\n", os.Args[1], oddOrEven(os.Args[1]))
  fmt.Printf("%s is %s\n", os.Args[2], oddOrEven(os.Args[2]))
  fmt.Printf("%s is %s\n", os.Args[3], oddOrEven(os.Args[3]))
  fmt.Printf("%s is %s\n", os.Args[4], oddOrEven(os.Args[4]))
}

func oddOrEven(value string) string {
        num, _ := strconv.Atoi(value)
        if num % 2 == 0 {
                return "even"
        } else {
                return "odd"
        }
}
```

go run main.go 12 78 99 37

12 is even

78 is even

99 is odd

37 is odd

Looping with the `for` statement

Looking at our program so far, it's obvious that although we're reusing code successfully by putting our odd/even detection logic in a function, we're still manually calling that function for each of our input parameters. Furthermore, we have hard-coded the reference to each input parameter. It would be nice to let the program accept as many or as few input parameters as our users feel inclined to provide and let it conduct our test on each of them.

In order to do that, we need to know the number of parameters passed in for each execution of the program and a way to process each of them in turn.

Getting the number of parameters is easy, because you can use the standard **len()** function on **Args** to return the number of parameters it holds:

```
numParams := len(os.Args)
```

So all we need now is a way to iterate through each item in **Args** and do something with it. Enter the **for** statement.

The **for** statement allows you to repeat a block of statements multiple times. If you've used any C-style programming language, then you're already familiar with it. The interesting thing is that, unlike most languages, the **for** statement is the only looping statement Go supports. However, it can be used in different ways to achieve the same thing as those other languages' **while** and **do-while** constructs.

The basic syntax of the **for** loop is as follows:

```
for init; condition; post {
    // run commands until condition is true
}
```

Where both the *init* and *post* statements are optional:

- The *init* statement is executed before the first iteration.
- The *condition* expression is evaluated before every iteration.
- The *post* statement is executed at the end of every iteration.

Note: Unlike with many C-style languages, there are no parentheses around the init, condition, *and* post *components of the* for *statement, and you must always include the braces (*{}*).*

Let's look at the various forms. First of all, an infinite loop:

```
for {
    // The statements in this block execute forever
}
```

This form is probably of limited use unless the entire program repeats forever. For example, games often run in a continuous loop, processing all the action and responding to user input.

You can also use this form to perform the equivalent of a **while**…**do** loop in other languages by adding a **break** statement to exit the loop when a condition is met:

```
total: = 1
for {
        if total > 1000 {
                break
        }
        total += 1
}
```

The following is the most common form and repeats the blocks in the statement from *init* until the *condition* is met, increasing the amount of *init* by the expression in *post* with each iteration:

```
for i := 1;  i <= 5; i++ {
        fmt.Printf("#%d\t", i)
}
```

It produces:

```
#1    #2    #3    #4    #5
```

The following replicates the **while** loop in other languages. It repeats the loop while the condition is met, but does not maintain its own loop counter like the previous example:

```
total := 1
for total < 1000 {
        total += total
}
```

The code in the **for** block repeats until **total** reaches 1,000.

For our purposes, the standard **for init; condition; post {...}** form will work well.

Code Listing 13

```
package main

import (
        "fmt"
        "os"
```

```go
        "strconv"
)

func main() {
        for i := 1; i < len(os.Args); i++ {
                fmt.Printf("%s is %s\n", os.Args[i],
                        oddOrEven(os.Args[i]))
        }
}

func oddOrEven(value string) string {
        num, _ := strconv.Atoi(value)
        if num % 2 == 0 {
                return "even"
        } else {
                return "odd"
        }
}
```

go run main.go 12 78 99 37

12 is even

78 is even

99 is odd

37 is odd

Note: By setting the initial loop counter i *to 1 and not zero, we avoid the first element in* Args *(the path to the calling program).*

The switch statement

Let's consider a variation on the application we are building so that we can demonstrate the other main conditional processing construct in Go: the **switch** statement.

In this example, we're going to allow the user to input a string. We'll parse the string and count the number of letters from **a** to **i**, from **j** to **r**, and from **s** to **z** and any other characters, and display the information to the user.

We already know how to accept multiple command-line parameters to our program, so allowing just one will be child's play!

We also know how to inspect the contents of a string. What's more, we know how to iterate through the string, character by character, with our trusty **for** statement.

The only thing we really need to consider is how to count the number of characters in each range.

We could use a succession of **if…else** statements, but that would get really ugly, very quickly:

```
if letter = "a" {

     // increment the appropriate total

} else if letter = "b" {

...

} else if letter = "c" {

     ...

} ...
```

…and so on.

A better approach would be to use a **switch** statement. A **switch** statement consists of (optionally) a condition to test, then several **case** clauses where we specify the conditions we are trying to match against and write the code to deal with those conditions. We can also provide a **default** clause that executes code that will run if none of the previous **case** conditions have been met.

For example:

```
switch operatingSystem := userOS; operatingSystem {

     case: "Windows"

          fmt.Println("Made by Microsoft")

     case: "OS X"

          fmt.Println("Made by Apple")

     case: "Linux"

          fmt.Println("Made by a lot of clever volunteers")

     default:

          fmt.Println("I don't recognize that operating system. GEEK!")

}
```

> *Note: Here we have not only demonstrated the* switch *statement, but a common Go idiom: the ability to include an initialization statement in a control structure. We can do the same in an* if *statement.*

If we just want to avoid a great big list of **if…else** statements, then we might not even need the condition:

```
numDaysInMonth := 30
```

```
switch {

      case numDaysInMonth >= 28 && numDaysInMonth <=29:

            fmt.Println("February")

      case numDaysInMonth == 30:

            fmt.Println("April, June, September, November")

      case numDaysInMonth == 31:

            fmt.Println("January, March, May, July, August, October,

                                                December")

}
```

Unlike most languages in which the code "falls through" to the next condition unless we provide a "break" statement at the end of each **case** code block, Go immediately exits the **switch** statement when a condition has been met and the code for the case has executed. If we want to continue checking subsequent conditions thereafter, we need to make this explicit through the use of the **fallthrough** keyword. This can be useful if we are testing data for which numerous conditions could apply.

But back to our example. We can use the **switch** statement to achieve our objective as follows:

Code Listing 14

```
package main

import (
      "fmt"
      "os"
)

func main() {

  numAtoI, numJtoR, numStoZ, numSpaces, numOther := 0, 0, 0, 0, 0
  sentence := os.Args[1]

  for i := 1; i < len(sentence); i++ {
    letter := string(sentence[i])
    switch letter {
      case "a", "b", "c", "d", "e", "f", "g", "h", "i":
numAtoI += 1
      case "j", "k", "l", "m", "n", "o", "p", "q", "r":
numJtoR += 1
      case "s", "t", "u", "v", "w", "x", "y", "z":
        numStoZ += 1
      case " ":
        numSpaces += 1
      default:
        numOther += 1
```

```
    }
  }

  fmt.Printf("A to I: %d\n", numAtoI)
  fmt.Printf("J to R: %d\n", numJtoR)
  fmt.Printf("S to Z: %d\n", numStoZ)
  fmt.Printf("Spaces: %d\n", numSpaces)
  fmt.Printf("Other: %d\n", numOther)
}
```

go run main.go "Everybody in the whole cell block (was dancing to the Jailhouse Rock)"

A to I: 25

J to R: 17

S to Z: 11

Spaces: 11

Other: 4

And we're done! All this should look pretty familiar to you by now. There's just one thing I'd like to point out: as we're looping through the letters in the input string, what we're getting back is a byte, not a string with a single character in it. That's why we need to use an explicit cast to convert that byte to a string:

letter := string(sentence[i])

This allows us to get the character itself rather than the byte that represents it.

For example, if we did a byte-by-byte extract from the string **Hello**, it would look like this:

72 101 108 108 111

and not what we want, which is this:

H e l l o

Chapter 7 Arrays, Slices, and Maps

In Chapter 5, we looked at the basic data "primitives" that Go supports: numbers, strings, and Booleans. As programmers, we often want to deal with several such values as a single unit. Go provides a number of built-in "composite" types to help us do this. In this chapter, we will consider arrays, slices, and maps.

Arrays

Arrays are a type of data structure that can store a fixed-size sequential collection of elements of the same type. The key thing to note here is that the length of an array is fixed. Once you have defined that, it will always hold exactly the same number of elements.

The way we declare an array in Go is to specify the name of the variable that holds the array, the number of elements that the array can store, and the data type of those elements. For example:

```go
var nums [6]int                     // holds 6 integers

var strings [3]string               // holds 3 strings

var preciseNums [10]float64         // holds 10 float64 numbers
```

We can retrieve the length of any array by using the `len()` function:

```go
fmt.Printf("Length: %d\n" + len(nums))    // Displays "Length: 6"
```

If we want to initialize the items in each array at the same time as we declare them, we can do so by specifying the value of each item in a comma-separated list within braces:

```go
totals := [5]int{1, 2, 3, 4, 5}
```

If you don't know the length of the array or you're just feeling lazy and want Go to work it out for you, you can use this notation:

```go
totals := [...]int{1, 2, 3, 4, 5}
```

If we don't initialize the array, each item in the array is initialized with the default value for its data type:

```go
fmt.Println(nums)                   // Displays "[0 0 0 0 0 0]"

fmt.Println(strings)                // Displays "[ ]"

fmt.Println(preciseNums)            // Displays "[0 0 0 0 0 0 0 0 0 0]"
```

We can set and get the value of individual array items by referring to their index positions. As with the position of bytes within strings, the numbering of Go array indexes begins at zero. So the second item in the array is at position 1, the third item in the array is at position 2, and so on.

```
nums[4] = 50                                // Sets the 5th item to 50

fmt.Println("nums[4] = " + nums[4])         // Displays "nums[4] = 50"
```

If we want to process each item in an array, we can use a **for** loop with the **range** keyword, specifying the name of a variable to store the individual array item during each loop iteration and the name of the array as a whole:

```
for _, item := range myArray {

        // Process the array item

}
```

Consider the following example, which loops through an array containing rainfall statistics for the previous five years and calculates the average (mean) rainfall for the period.

Code Listing 15

```
package main

import (
        "fmt"
)

func main() {

        total := 0
        mean := 0
        rainfallStats := [5]int{1091, 2010, 995, 1101, 1111}
        for _, value := range rainfallStats {
                total += value
        }
        mean = total / len(rainfallStats)
        fmt.Printf("Average rainfall: %d mm\n", mean)
}
```

Average rainfall: 1261 mm

Array types are one-dimensional, but you can combine array types to build multi-dimensional data structures. The following example creates an array with five rows and two columns, and outputs the value stored in each row:

Code Listing 16

```
package main

import "fmt"

func main() {
    // Array with 5 rows and 2 columns
```

```
    var arr = [5][2]int{ {0,0}, {2,4}, {1,3}, {5,7}, {6,8}}

    // Display each array element's value
    for i := 0; i < 5; i++ {
        for j := 0; j < 2; j++ {
            fmt.Printf("arr[%d][%d] = %d\n", i,j, arr[i][j] )
        }
    }
}
```

arr[0][0] = 0

arr[0][1] = 0

arr[1][0] = 2

arr[1][1] = 4

arr[2][0] = 1

arr[2][1] = 3

arr[3][0] = 5

arr[3][1] = 7

arr[4][0] = 6

arr[4][1] = 8

Initializing big arrays in a single line of does not make your code easy to read, so Go allows you to put the elements on successive lines, as follows:

```
a := [5]int{
    93,
    79,
    34,
    202,
    17,
}
```

Notice the last comma? Most languages would complain about that. Go actually *requires* it. Why? Because it makes it easier to comment out items that you no longer require without having to adjust the position of the last comma:

```
a := [5]int{
    93,
    79,
    34,
```

```
        //202,
    17,
}
```

You've got to love Go—it's always trying to make your life as a programmer easier!

However, we now have an array with space for five items that only contains four. If we print this out, we can see that Go has moved everything up and substituted a default value for the empty item in the list:

```
fmt.Println(a)    // Displays "[93 79 34 17 0]"
```

This is not necessarily what we want, and one of several reasons that you don't often see arrays being used in Go programs. They can be a bit of a pain to work with, especially when it comes to resizing them—you can't! The only way around that is to create a new array.

> 💡 **Tip: Make sure you really want an array before you use one. Arrays are limited, and don't have as much flexibility as their much more flexible cousins the slices, described later in this chapter.**

Another thing to note about arrays is that when you pass one into a function as a parameter, what the function receives is not the array itself, but a *copy* of the array. Any changes you make to the array in the function will not affect the original array. This is known as *passing by value*, and can be computationally expensive if you are dealing with a very large array.

The following example demonstrates this:

Code Listing 17

```go
package main

import (
        "fmt"
)

func main() {
        myArray := [...]string{"Apples", "Oranges", "Bananas"}
        fmt.Printf("Initial array values: %v\n", myArray)
        myFunction(myArray)
        fmt.Printf("Final array values: %v\n", myArray)
}

func myFunction(arr [3]string) {
        // Change Oranges to Strawberries
        arr[1] = "Strawberries"
        fmt.Printf("Array values in myFunction(): %v\n", arr)
}
```

```
Initial array values: [Apples Oranges Bananas]

Array values in myFunction(): [Apples Strawberries Bananas]

Final array values: [Apples Oranges Bananas]
```

Like other languages, Go lets you pass by value or by *reference*. The difference is that when you pass by reference, you pass a *pointer* to the memory location where the value is stored, so that you are working with the original object. If the function changes the value of that object, the changes are permanent once the function has finished executing. We'll talk about pointers in Chapter 9 when we discuss user-defined types, but bear this in mind when choosing to use an array in Go.

So, you can now see that the array type has its limitations. That's not to say we've wasted time covering it here, because it leads us into talking about another, more capable composite type based on the array, called a *slice*.

Slices

Slices are a key data type in Go. They do everything you expect an array to do, and more.

A slice is basically a "view" into an underlying array.

You can create and populate a slice simply by defining an array without specifying the size:

```
fruits := []string{"Apples", "Oranges", "Bananas", "Kiwis"}
```

This approach provides several benefits.

One is that you can refer to a range of items in the slice using a similar notation as you used with substrings in Chapter 5—effectively taking a "subslice" from the slice.

The syntax to use is:

```
[lower-bound, upper-bound]
```

… where **lower-bound** is included in the **subslice** and **upper-bound** is excluded from it.

For example:

Code Listing 18

```go
package main

import (
        "fmt"
)

func main() {
        fruits := [...]string{"apples", "oranges", "bananas", "kiwis"}
        fmt.Printf("%v\n", fruits[1:3])
```

```
        fmt.Printf("%v\n", fruits[0:2])
        fmt.Printf("%v\n", fruits[:3])
        fmt.Printf("%v\n", fruits[2:])
}
```

[oranges bananas]

[apples oranges]

[apples oranges bananas]

[bananas kiwis]

Another benefit is that when you pass a slice to a function as a parameter, instead of the function receiving a *copy* of the underlying array it gets a *pointer* to it. That means anything you do to the slice within the function is reflected in the underlying array. We can demonstrate this by rewriting the program in Code Listing 17 to use a slice instead of an array.

Code Listing 19

```
package main

import (
        "fmt"
)

func main() {
        mySlice := []string{"Apples", "Oranges", "Bananas"}
        fmt.Printf("Initial slice values: %v\n", mySlice)
        myFunction(mySlice)
        fmt.Printf("Final slice values: %v\n", mySlice)
}

func myFunction(fruits []string) {
        // Change Oranges to Strawberries
        fruits[1] = "Strawberries"
        fmt.Printf("Slice values in myFunction(): %v\n", fruits)
}
```

Initial slice values: [Apples Oranges Bananas]

Slice values in myFunction(): [Apples Strawberries Bananas]

Final slice values: [Apples Strawberries Bananas]

So far we have created slices directly from the items we want to store in them. If you want to create a new slice and then add data to it later, you need to use the built-in **make()** function.

mySlice := make([]int)

If you know the initial size, you can specify it as follows:

```
mySlice := make([]int, 4)
```

You can also specify a maximum size by supplying a third parameter to the **make()** function and retrieve its value at any time by calling **cap()**. The following code creates a slice that will store up to eight **int** values, with an initial capacity of four such values:

Code Listing 20

```go
package main

import (
        "fmt"
)

func main() {
        mySlice := make([]int, 4, 8)
        fmt.Printf("Initial Length: %d\n", len(mySlice))
        fmt.Printf("Capacity: %d\n", cap(mySlice))
        fmt.Printf("Contents: %v\n", mySlice)
}
```

```
Initial Length: 4
```

```
Capacity: 8
```

```
Contents: [0 0 0 0]
```

If we only know the maximum number of items our slice should be able to hold (eight), but not the number of initial values, we could create it like this:

```
mySlice := make([]int, 0, 8)
```

Let's populate our slice:

```
mySlice[0] = 1
```

```
mySlice[1] = 3
```

```
mySlice[2] = 5
```

```
mySlice[3] = 7
```

That's a rather cumbersome way of doing it, so let's use **append()** instead:

```
mySlice = append(mySlice, 1, 3, 5, 7)
```

The main benefit of a slice is that it can be resized dynamically. Let's say we have specified a capacity for the slice, but we've added more items than the capacity allows. What happens?

Let's see:

```go
package main

import (
        "fmt"
)

func main() {

        mySlice := make([]int, 0, 8)
        mySlice = append(mySlice, 1, 3, 5, 7, 9, 11, 13, 17)

        fmt.Printf("Contents: %v\n", mySlice)
        fmt.Printf("Number of Items: %d\n", len(mySlice))
        fmt.Printf("Capacity: %d\n", cap(mySlice))

        mySlice = append(mySlice, 19)

        fmt.Printf("Contents: %v\n", mySlice)
        fmt.Printf("Number of Items: %d\n", len(mySlice))
        fmt.Printf("Capacity: %d\n", cap(mySlice))

}
```

Contents: [1 3 5 7 9 11 13 17]

Number of Items: 8

Capacity: 8

Contents: [1 3 5 7 9 11 13 17 19]

Number of Items: 9

Capacity: 16

What happened here is that when we appended the ninth item to the slice, Go automatically doubled the capacity for us in order to make extra room in case we want to continue adding items. So the capacity we specified when we created the slice is not treated as a hard limit, just a guideline for Go to determine the initial memory allocation.

However, this only works because we used the slice-specific **append()** function. If we had attempted to add the ninth integer using the standard array approach, we would receive a run-time error:

Code Listing 22

```go
package main
```

```go
import (
    "fmt"
)

func main() {

    mySlice := make([]int, 0, 8)
    mySlice = append(mySlice, 1, 3, 5, 7, 9, 11, 13, 17)

    fmt.Printf("Contents: %v\n", mySlice)
    fmt.Printf("Number of Items: %d\n", len(mySlice))
    fmt.Printf("Capacity: %d\n", cap(mySlice))


    mySlice[8] = 19

    fmt.Printf("Contents: %v\n", mySlice)
    fmt.Printf("Number of Items: %d\n", len(mySlice))
    fmt.Printf("Capacity: %d\n", cap(mySlice))


}
```

Contents: [1 3 5 7 9 11 13 17]

Number of Items: 8

Capacity: 8

panic: runtime error: index out of range

goroutine 1 [running]:

panic(0xda900, 0xc82000a080)

   /usr/local/go/src/runtime/panic.go:464 +0x3e6

main.main()

   .../src/hello/main.go:17 +0x725

exit status 2

This is because index position 8 does not exist. It's only by using **append()** that Go can detect that we need extra capacity and then provide it.

Finally, I mentioned that one of the benefits of slices is that you can pass them into functions as pointers to their underlying arrays and therefore make changes to the original (and not just a copy), which is what happens when you pass an array to a function.

But what if you genuinely want to take a copy of a slice? You can do that with Go's **copy()** function:

```go
package main

import (
    "fmt"
)

func main() {
    mySlice := make([]int, 0, 8)
    mySlice = append(mySlice, 1, 3, 5, 7, 9, 11, 13, 17)

    mySliceCopy := make([]int, 8)
    copy(mySliceCopy, mySlice)

    mySliceCopy[3] = 999
    fmt.Printf("mySlice: %v\n", mySlice)
    fmt.Printf("mySliceCopy: %v\n", mySliceCopy)
}
```

mySlice: [1 3 5 7 9 11 13 17]

mySliceCopy: [1 3 5 999 9 11 13 17]

Hopefully you can now see some of the immense advantages there are to working with slices compared to arrays. For these reasons, it's actually quite rare to see arrays in Go: slices are almost always the best choice.

Maps

A map is an unordered collection of key-value pairs. If you've been programming for a while, you've seen this kind of structure referred to elsewhere as an associative array, a dictionary, or a hash.

The idea behind a map is that you access its values by referencing the key that points to each value. The keys can be any data type that you can test for equality. So strings, integers, floating-point numbers, and so on are all good, but you cannot use an array as a key, for example.

Maps are often used as a kind of in-memory "lookup" table. Say for instance that we want to keep a record of famous actors and their ages. We can create a map using the **make()** function, just as we did with a slice:

actor := make(map[string]int)

In the **map** declaration, the type within square brackets is the type of data we want to use for the keys, and the type following it is the data type of the values.

We then assign an item to the map using the following syntax:

```
actor["Redford"] = 79
```

We then retrieve it by specifying the key of the value we want to access:

```
fmt.Println(actor["Redford"])
```

Code Listing 24

```go
package main

import "fmt"

func main() {
    actor := make(map[string]int)
    actor["Paltrow"] = 43
    actor["Cruise"] = 53
    actor["Redford"] = 79
    actor["Diaz"] = 43
    actor["Kilmer"] = 56
    actor["Pacino"] = 75
    actor["Ryder"] = 44

    fmt.Printf("Robert Redford is %d years old\n",
                                        actor["Redford"])
    fmt.Printf("Cameron Diaz is %d years old\n", actor["Diaz"])
    fmt.Printf("Val Kilmer is %d years old\n", actor["Kilmer"])
}
```

Robert Redford is 79 years old

Cameron Diaz is 43 years old

Val Kilmer is 56 years old

Generously, Go also provides us with a shorthand syntax for creating maps:

Code Listing 25

```go
package main

import "fmt"

func main() {
    actor := map[string]int{
        "Paltrow": 43,
        "Cruise":  53,
        "Redford": 79,
        "Diaz":    43,
        "Kilmer":  56,
        "Pacino":  75,
```

```
        "Ryder":     44,
    }

    fmt.Printf("Robert Redford is %d years old\n",
                                    actor["Redford"])
    fmt.Printf("Cameron Diaz is %d years old\n", actor["Diaz"])
    fmt.Printf("Val Kilmer is %d years old\n", actor["Kilmer"])
}
```

Robert Redford is 79 years old

Cameron Diaz is 43 years old

Val Kilmer is 56 years old

If we try to access a map value with a nonexistent key, Go will return the default value for the data type:

fmt.Printf("Anthony Hopkins is %d years old\n", actor["Hopkins"])

This results in:

Anthony Hopkins is 0 years old

In our example, that's unlikely to cause much confusion: we can simply test for zero to see if our users have specified a valid key. But what if zero is a legitimate value? How can we differentiate between that and an invalid key?

The solution is to use the "comma OK" syntax. This takes advantage of the map's ability to return two values instead of one when we attempt to retrieve an item. The first (**age**) is the value itself, and the second (**ok**) is a Boolean that tells us whether or not the key used was valid:

if age, ok := actor["Hopkins"]; ok {

    fmt.Printf("Anthony Hopkins is %d years old\n", age)

} else {

    fmt.Println("Actor not recognized.")

}

💡 *Tip: You can use an initialization statement to declare a variable in the same line as an* if *statement. This leads to clean, understandable code. The variable's scope is the statement in which it is defined.*

You can iterate through a map using a **for** loop with the **range** statement, just as we did with an array. Note, however, that the items in a map will be unordered, so you cannot rely on them being in the order you inserted them, or indeed in the same order each time you loop through them. This is by design: the Go team didn't want programmers to rely upon an ordering that was essentially unreliable, so they randomized the iteration order to make that impossible.

```go
package main

import "fmt"

func main() {
       actor := map[string]int{
               "Paltrow": 43,
               "Cruise":  53,
               "Redford": 79,
               "Diaz":    43,
               "Kilmer":  56,
               "Pacino":  75,
               "Ryder":   44,
       }

       for i := 1; i < 4; i++ {
               fmt.Printf("\nRUN NUMBER %d\n", i)
               for key, value := range actor {
                       fmt.Printf("%s : %d years old\n", key, value)
               }
       }
}
```

RUN NUMBER 1
Ryder : 44 years old
Paltrow : 43 years old
Cruise : 53 years old
Redford : 79 years old
Diaz : 43 years old
Kilmer : 56 years old
Pacino : 75 years old

RUN NUMBER 2
Kilmer : 56 years old
Pacino : 75 years old
Ryder : 44 years old
Paltrow : 43 years old
Cruise : 53 years old
Redford : 79 years old
Diaz : 43 years old

RUN NUMBER 3
Paltrow : 43 years old
Cruise : 53 years old
Redford : 79 years old
Diaz : 43 years old
Kilmer : 56 years old

```
Pacino : 75 years old
Ryder : 44 years old
```

If you want to sort the items in a map, you must rely on another structure for help. The following example demonstrates pulling the **actor** keys into a slice, ordering the slice using the **sort** package, then iterating through the slice to retrieve the actor names from the map in alphabetical order.

Code Listing 27

```go
package main

import (
        "fmt"
        "sort"
)

func main() {
        actor := map[string]int{
                "Paltrow": 43,
                "Cruise":  53,
                "Redford": 79,
                "Diaz":    43,
                "Kilmer":  56,
                "Pacino":  75,
                "Ryder":   44,
        }

        // Store the keys in a slice
        var sortedActor []string
        for key := range actor {
                sortedActor = append(sortedActor, key)
        }
        // Sort the slice alphabetically
        sort.Strings(sortedActor)

        /* Retrieve the keys from the slice and use
           them to look up the map values */
        for _, name := range sortedActor {
                fmt.Printf("%s : %d years old\n", name, actor[name])
        }
}
```

```
Cruise : 53 years old

Diaz : 43 years old

Kilmer : 56 years old

Pacino : 75 years old
```

Paltrow : 43 years old

Redford : 79 years old

Ryder : 44 years old

Chapter 8 User-Defined Types

As programmers, we often must represent real-world objects in code, and being able to store all the information we need about a given "thing" in a discrete entity, along with the operations we need to perform on that information, is very helpful. This is the essence of object-oriented programming, a paradigm based on the concept of "objects" that store data (state) in the form of *fields*, and code (behavior) that can access and manipulate that data in the form of *methods.*

Go supports types, methods, and interfaces that enable us to work in an object-oriented fashion. You might argue that Go isn't strictly an object-oriented language in that it does not directly support a type hierarchy, but it goes a long way towards implementing many of the features you will find in a true object-oriented language.

A word on pointers

Before we dive into how Go implements all of this, we need to discuss pointers. The reason for doing so will become apparent as we move forward.

If you've never used a language like C or C++, the concept of a pointer might be unfamiliar to you. In programming parlance, a pointer contains the memory address of a variable.

Here is how we define a pointer to a variable of type **int**:

var p *int

And this is how we assign the memory address of the variable **q** to pointer **p**:

q = &i

To display the pointer's underlying value, use the ***** (indirection or de-referencing) operator:

*p = 123   // De-reference pointer p to get i and assign a value to it

fmt.Println(*p)   // Print the value of i by dereferencing p

The usage of pointers is demonstrated in the following example:

Code Listing 28

```
package main

import "fmt"

func main() {
    a, b := 20, 71
```

```
      // pointer to a
      p := &a
      // read a via the pointer
      fmt.Printf("Value of a is: %d\n", *p)
      // set a via the pointer
      *p = 21
      // display the new value of a
      fmt.Printf("New value of a is: %d\n", a)

      // pointer to b
      p = &b
      // read b via the pointer
      fmt.Printf("Value of b is: %d\n", *p)
      // add 10 to b via its pointer
      *p = *p + 10
      // display the new value of b
      fmt.Printf("New value of b is: %d\n", b)
}
```

Value of a is: 20

New value of a is: 21

Value of b is: 71

New value of b is: 81

Structs

Go doesn't use the word "object." What you might think of as an object in another language is a **struct** in Go.

To define a **struct**, use the **type** keyword followed by an identifier and then the **struct** keyword. Add any fields you want to describe the **struct**'s state.

The following example defines a structure for a rectangle called **rect**, sets initial values for its **height** and **width** properties, and then displays them to the user.

Code Listing 29

```
package main

import (
      "fmt"
)

type rect struct {
      height int
```

```go
        width   int
}

func main() {
        r := rect{height: 12, width: 20}
        fmt.Printf("Height: %d\nWidth: %d\n", r.height, r.width)
}
```

Height: 12

Width: 20

Note how we access the fields on the **struct** using dot notation in the form:

struct_name.field

What this is really doing behind the scenes is de-referencing the pointer to the **struct** for us and then accessing its field, so if we really wanted to, we could write:

(*struct_name).field

If you don't explicitly assign values to a field, Go automatically assigns the default value for the data type.

> 📝 ***Note: As we saw in Chapter 3, if you capitalize the first letter of a variable name, that variable becomes global. The same concept applies to fields within structures. Other programming languages use keywords like*** public, private, package, protected, ***and so on to allow the developer to define the visibility and accessibility of variables within different contexts. Go's approach is nothing short of genius, in my humble opinion!***

We used the following syntax in Code Listing 29 to define the **struct** and initialize its fields:

r := rect{height: 12, width: 20}

To get a pointer to the actual structure data in memory, use the **&** operator:

myRectangle := &rect{

      height: 12,

      width: 20,

}

If we know the order the fields are defined in, we can assign values to those fields in the following way:

r := rect{12, 20}

We can also assign field values using dot notation:

```
r.height = 12
r.width = 20
```

To create an instance of a structure and return a pointer to it, use the following syntax:

```
var myRectangle *rect
myRectangle = new(rect)
```

This is quite verbose, so Go provides a shorthand:

```
myRectangle := new(rect)
```

Methods

We create methods in Go by defining a function and then binding it to a **struct**. We need to decide whether our method needs to act on the actual structure itself or just a copy of it. For example, if our method needs to change the values of fields within the structure, we would need access to the original structure because if we worked on a copy, the original field values would be unaffected by any change. Another good reason for using a pointer is in case the structure is large and would therefore be expensive to copy.

Let's create a method that calculates the area of our rectangle and binds it to the **rect** structure:

Code Listing 30

```go
package main

import (
        "fmt"
)

type rect struct {
        height int
        width  int
}

func main() {
        r := rect{height: 12, width: 20}
        fmt.Printf("Height: %d\n", r.height)
        fmt.Printf("Width: %d\n", r.width)
        fmt.Printf("Area: %d\n", r.area())
}

func (r rect) area() int {
        h := r.height
        w := r.width
```

```
        return h * w
}
```

Height: 12

Width: 20

Area: 240

The binding occurs by specifying the *receiver* of the method in the parentheses immediately following the **func** keyword:

```
func (r rect) area() int {

        // Code to calculate the area

}
```

In this instance, we don't need to modify the field values, so we are working on a copy of the structure rather than the original.

Let's create another function called **double()** that doubles the dimensions of our rectangle.

Code Listing 31

```go
package main

import (
        "fmt"
)

type rect struct {
        height int
        width  int
}

func main() {
        r := rect{height: 12, width: 20}
        fmt.Printf("Height: %d\n", r.height)
        fmt.Printf("Width: %d\n", r.width)
        fmt.Printf("Area: %d\n", r.area())

        r.double()
        fmt.Printf("Height: %d\n", r.height)
        fmt.Printf("Width: %d\n", r.width)
        fmt.Printf("Area: %d\n", r.area())

}

func (r rect) area() int {
        h := r.height
```

```
        w := r.width
        return h * w
}

func (r rect) double() {
        r.height *= 2
        r.width *=2
}
```

Height: 12

Width: 20

Area: 240

Double it!

Height: 12

Width: 20

Area: 240

That's not what we intended! In this case, the fact that we are working on a copy of the structure and not the original renders our function virtually useless. Let's amend the **double()** method so that the receiver is a pointer to the original **rect** structure.

Code Listing 32

```
package main

import (
        "fmt"
)

type rect struct {
        height int
        width  int
}

func main() {
        r := rect{height: 12, width: 20}
        fmt.Printf("Height: %d\n", r.height)
        fmt.Printf("Width: %d\n", r.width)
        fmt.Printf("Area: %d\n", r.area())

        fmt.Printf("\nDouble it!\n\n")
        r.double()
```

```go
        fmt.Printf("Height: %d\n", r.height)
        fmt.Printf("Width: %d\n", r.width)
        fmt.Printf("Area: %d\n", r.area())

}

func (r rect) area() int {
        h := r.height
        w := r.width
        return h * w
}

func (r *rect) double() {
        r.height *= 2
        r.width *=2
}
```

Height: 12

Width: 20

Area: 240

Double it!

Height: 24

Width: 40

Area: 960

The **double()** method is now working as intended.

Embedded types

Types can contain other types. When you embed one type in another, the embedded type is anonymous: you don't give it a name. Instead, you refer to it via the name of its *type*.

For example, here is a type that describes a sale discount:

```go
type Discount struct {
    percent    float32
    promotionId string
}
```

Here is another type that reflects a special discount with extra off the discounted sale price:

```
type ManagersSpecial struct {
      Discount     // The embedded type
      extraoff float32
}
```

We can instantiate these types as follows:

```
januarySale := Discount{15.00, "January"}
managerSpecial := ManagersSpecial{januarySale, 10.00}
```

We can then refer to the fields on the embedded type using the type name as follows:

```
managerSpecial.Discount.percent          // "15.00
managerSpecial.Discount.promotionId      // "January"
```

If we bind a method to **Discount**, we can invoke it using the same syntax:

```
managerSpecial.Discount.someMethod(someParameter)
```

The following sample shows this concept in action:

Code Listing 33

```go
package main

import (
      "fmt"
)

type Discount struct {
      percent     float32
      promotionId string
}

type ManagersSpecial struct {
      Discount
      extraoff float32
}

func main() {

      normalPrice := float32(99.99)

      januarySale := Discount{15.00, "January"}
      managerSpecial := ManagersSpecial{januarySale, 10.00}
```

```
        discountedPrice := januarySale.Calculate(normalPrice)
        managerDiscount := managerSpecial.Calculate(normalPrice)

        fmt.Printf("Original price: $%4.2f\n", normalPrice)
        fmt.Printf("Discount percentage: %2.2f\n",
                                            januarySale.percent)
        fmt.Printf("Discounted price: $%4.2f\n", discountedPrice)
        fmt.Printf("Manager's special: $%4.2f\n", managerDiscount)
}

func (d Discount) Calculate(originalPrice float32) float32 {
        return originalPrice - (originalPrice / 100 * d.percent)
}

func (ms ManagersSpecial) Calculate(originalPrice float32) float32 {
        return ms.Discount.Calculate(originalPrice) - ms.extraoff
}
```

Original price: $99.99

Discount percentage: 15.00

Discounted price: $84.99

Manager's special: $74.99

Interfaces

A Go interface is a contract that defines the methods that any type using the interface must implement and expose.

For example, say we have defined types for different vehicles: a car, a bicycle, and a motorcycle. Each of these vehicles can accelerate, decelerate, and brake. The actual details of how each of these actions is achieved is the responsibility of the type, but the fact that they must all accommodate these behaviors can be enforced by an interface.

The benefit of this approach is that once a number of types implement the same interface, they can all be regarded as being equivalent in some way. To continue our example, even though we have cars, cycles, and motorcycles, if our program simply wants to drive them, we can use the same **drive()** function, pass in *any* vehicle, and **drive()** will call the appropriate type-specific **accelerate()**, **decelerate()**, and **brake()** methods.

This is really useful, but is hard to explain without a concrete example, so let's create a very simple (and rather contrived) example to demonstrate the concept.

I'm going to define two types: one for animals and one for people. The **Animal** class has a **sound** field, and the **Person** class has a **name** field. Both types need to be able to say hello. The person will introduce him or herself, and the animal will make an appropriate noise depending on what type of animal it is.

In order to enforce this behavior, I'm going to require that both types have a method called **sayHi()**. I can do this by defining an interface called **Greeter** that specifies the method signature of **sayHi()**. Any type that implements the interface must have a **sayHi()** method with the exact same signature.

```
type Greeter interface {

        sayHi() string

}
type Person struct {

        Name string

}
type Animal struct {

        Sound string

}
func (p Person) SayHi() string {

        return "Hello! My name is " + p.Name

}


func (a Animal) SayHi() string {

        return a.Sound

}
```

I can then create a function called **greet()** that accepts any type that implements the **Greeter** interface (a person or an animal) and calls the appropriate **sayHi()** method for the type:

```
func greet(i Greeter) {

        fmt.Println(i.SayHi())

}
```

Even though we're supplying different types, the **greet()** function can deal with all of them. This is extremely powerful.

Here is the full code and output:

Code Listing 34

```
package main

import (
        "fmt"
)
```

```go
type Person struct {
      Name string
}

type Animal struct {
      Sound string
}

type Greeter interface {
      SayHi() string
}

// this method is specific to the Person class
func (p Person) SayHi() string {
      return "Hello! My name is " + p.Name
}

// this method is specific to the Animal class
func (a Animal) SayHi() string {
      return a.Sound
}

/* this method can be called on any type that
implements the Greeter interface */
func greet(i Greeter) {
      fmt.Println(i.SayHi())
}

func main() {
      man := Person{Name: "Bob Smith"}
      dog := Animal{Sound: "Woof! Woof!"}

      fmt.Println("\nPerson : ")
      greet(man)

      fmt.Println("\nAnimal : ")
      greet(dog)

}
```

Person :

Hello! My name is Bob Smith

Animal :

Woof! Woof!

The empty interface

The empty interface looks like this:

interface {}

It's an interface that all types implement—basically because there is nothing to implement! Other languages like .NET and Java refer to this type of interface as a "marker interface."

This might seem like nonsense, but it can actually be very useful, and in fact it is used widely in the Go standard libraries. The reason it is useful is that it gives you a way to express an unknown type that is determined at runtime.

Because Go is a statically typed language, the compiler complains very loudly if any variable has not been assigned a type, so the empty interface helps us get around this restriction.

Think of **interface{}** as being a bit like **Object** in other languages.

Here is an example of how we might use the empty interface.

Code Listing 35

```go
package main

import (
        "fmt"
)

func main() {
        displayType(42)
        displayType(3.14)
        displayType("ここでは文字列です")
}

func displayType(i interface{}) {
        switch theType := i.(type) {
        case int:
                fmt.Printf("%d is an integer\n", theType)
        case float64:
                fmt.Printf("%f is a 64-bit float\n", theType)
        case string:
                fmt.Printf("%s is a string\n", theType)
        default:
                fmt.Printf("I don't know what %v is\n", theType)
        }
}
```

42 is an integer

3.140000 is a 64-bit float

ここでは文字列です is a string

The function **displayType()** accepts any data type (because of the empty interface parameter). It then uses a *type switch* to return the actual type of the parameter that was passed into it.

Type assertion

If you have used the empty interface to access a value of an unknown data type and want to convert it to a known type so that you can work with it, you cannot simply use a type conversion because the empty interface is not a type. Instead, you must use a *type assertion*.

A type assertion takes the following form:

var anything interface{} = "something"

aString := anything.(string)

Code Listing 36

```go
package main

import (
        "fmt"
)

func main() {
        var anything interface{} = "something"
        aString := anything.(string)
        fmt.Println(aString)
}
```

something

If Go is unable to perform the conversion, it will "panic."

Code Listing 37

```go
package main

import (
        "fmt"
)

func main() {
        var anything interface{} = "something"
        aInt := anything.(int)
        fmt.Println(aInt)
}
```

```
panic: interface conversion: interface is string, not int
goroutine 1 [running]:
panic(0xda640, 0xc820010180)
        /usr/local/go/src/runtime/panic.go:464 +0x3e6
main.main()
        .../src/hello/main.go:9 +0xa8
exit status 2
```

To protect against this eventuality, take advantage of the fact that the assertion can pass back two parameters: one containing the converted type and another indicating success or failure. You can then test the second parameter and gracefully deal with any errors.

Code Listing 38

```go
package main

import (
        "fmt"
)

func main() {
        var anything interface{} = "something"
        aInt, ok := anything.(int)
        if !ok {
                fmt.Println("Cannot turn input into an integer")
        } else {
                fmt.Println(aInt)
        }
}
```

```
Cannot turn input into an integer
```

Chapter 9 Concurrency

One of Go's most exciting features is its fantastic support for concurrency, which allows programs written in the language to scale massively. In this chapter, we'll define concurrency and look at the two main features of the language that make concurrent programming easy in Go: goroutines and gochannels.

Processes, threads, and concurrency

When a program executes, the operating system creates a *process* for it to run in. This process acts as a container for the program and all the resources it uses while it executes. These resources include things like the memory the program accesses, any devices it uses, the file handles it uses for I/O, and threads.

A *thread* is the smallest unit of processing that can be performed in an operating system. A single process can contain multiple threads. Whereas a program that is single-threaded executes each of its instructions sequentially until the last instruction is reached, one with multiple threads allows multiple operations to run seemingly at the same time. For example, consider a program that allows you to watch a video while downloading it.

This delegation of operations to multiple threads of execution is known as *concurrency*. The dictionary definition of concurrent is "occurring or existing simultaneously," and concurrency in computer programming means exactly the same thing: the ability of a program to use multiple threads in order to get more done in less time.

Goroutines

A goroutine is a method or function that can run concurrently with other functions. Note that any Go program automatically consists of one goroutine: the program itself.

To create a new goroutine, we need only to prefix the call to a method or function with the **go** keyword:

```
go myFunction(parameter)
```

Because goroutines are extremely lightweight, there is nothing to stop us from creating hundreds or even thousands of them. Each executes and immediately returns control to the next line in the program without the program waiting for whatever function was called in the goroutine to complete.

Consider the following example:

```go
package main

import (
        "fmt"
        "time"
)

func message(s string) {
        for i := 0; i < 5; i++ {
                time.Sleep(100 * time.Millisecond)
                fmt.Println(s)
        }
}

func main() {
        go message("goroutine")
        message("normal")
}
```

This program has a function called **message()** that accepts a string value and prints it out five times. The **message()** function is called once as a goroutine running in its own thread, and once as normal within the main program thread.

When you run the program, you can see the two different threads competing to display their results:

goroutine

normal

goroutine

normal

goroutine

normal

normal

goroutine

goroutine

normal

The order in which the threads are processed is indeterminate, so when you run the program again, you might get a completely different result:

normal

goroutine

goroutine

normal

normal

goroutine

goroutine

normal

normal

goroutine

Note the use of the Go standard library **time** package to introduce a delay in the **message()** function. Without this delay, the program would call the goroutine and immediately drop down into the standard function call, and the program would terminate before the goroutine had a chance to finish.

Code Listing 40

```go
package main

import (
        "fmt"
)

func message(s string) {
        for i := 0; i < 5; i++ {
                fmt.Println(s)
        }
}

func main() {
        go message("goroutine")
        message("normal")
}
```

normal

normal

normal

normal

normal

Gochannels

Gochannels are the plumbing that allow different goroutines to communicate with each other. You can use a channel to send a value from one goroutine and receive it in another.

You must create a channel before you attempt to use it. You create a channel with **make()** like you do with slices and maps. The syntax to create a channel is as follows:

myChannel := make(chan *type*)

A channel can only accept values of any one type, so you need to specify the type when creating it.

You can put a value into the channel with the following syntax:

myChannel <- *value*

You retrieve a value from the channel like so:

myVariable: <- myChannel

By default, sends and receives are blocked until the sender and receiver are both ready. This allows goroutines to synchronize without explicit locks or condition variables.

Let's illustrate the use of channels with a programmatic implementation of a "numbers station."

A numbers station is a peculiar artifact of the Cold War. During this period, shortwave radio enthusiasts started to notice bizarre radio broadcasts. They would start with a weird melody or a series of beeps, and then a strange woman's or a child's voice would start announcing a series of seemingly random numbers, presumably representing some unknown code to communicate with agents in the field.

Weird, and not a little creepy.

Anyway, we'll create a function that generates a series of these numbers and drops them into a channel, which our program can then access.

Code Listing 41

```go
package main

import (
        "fmt"
        "math/rand"
        "time"
)

func broadcast(c chan int) {
        // infinite loop to create random numbers
        for {
                /* generate a random number 0-999
                and put it into the channel */
                c <- rand.Intn(999)
        }
}
```

```
func main() {
        numbersStation := make(chan int)

        // execute broadcast in a separate thread
        go broadcast(numbersStation)

        // retrieve values from the channel
        for num := range numbersStation {
                // delay for artistic effect only
                time.Sleep(1000 * time.Millisecond)
                fmt.Printf("%d ", num)
        }
}
```

878 636 407 983 895 735 520 998 904 150 212 538 750 362 436 215 630 506 20 914 272 207 266 298 135 565 43 964 942 705 562 249 734 203 840 152 357 718 84 189 871 256 ...

Let's break that down.

The **broadcast()** function accepts an integer channel **c** as a parameter. It then creates an infinite number of random integers between zero and 999 by using another Go package we haven't seen yet: **math/rnd**, and drops those into the channel.

In **main()**, we're creating a new integer channel called **numbersStation**. We then call **broadcast()** as a goroutine in a separate thread, which then continually generates random numbers and drops them into the channel.

Our **for** loop accesses the channel and displays each of the random numbers generated by the broadcast one at a time, with a one-second delay between each. This all happens so quickly that if we didn't have a delay, the output would just be a blur of numbers—far too many for our number station operator to read out.

Let's create a slightly more useful example—one that creates new account numbers for a fictitious banking application. The account number will be eight digits, starting from 30000001.

Code Listing 42

```
package main

import (
        "fmt"
)

func generateAccountNumber(accountNumberChannel chan int) {
        // internal variable to store last generated account number
        var accountNumber int
        accountNumber = 30000001
```

```go
        for {
                accountNumberChannel <- accountNumber
                // increment the account number by 1
                accountNumber += 1
        }
}

func main() {
        accountNumberChannel := make(chan int)
        // start the goroutine that generates account numbers
        go generateAccountNumber(accountNumberChannel)

        fmt.Printf("SMITH: %d\n", <-accountNumberChannel)
        fmt.Printf("SINGH: %d\n", <-accountNumberChannel)
        fmt.Printf("JONES: %d\n", <-accountNumberChannel)
        fmt.Printf("LOPEZ: %d\n", <-accountNumberChannel)
        fmt.Printf("CLARK: %d\n", <-accountNumberChannel)
}
```

SMITH: 30000001

SINGH: 30000002

JONES: 30000003

LOPEZ: 30000004

CLARK: 30000005

If we modify the code in the previous listing so that we close the channel after five account numbers have been requested, then any subsequent read of the channel will return **nil**. We can check for this eventuality using the "comma OK" syntax.

Code Listing 43

```go
package main

import (
        "fmt"
)

func generateAccountNumber(accountNumberChannel chan int) {
        // internal variable to store last generated account number
        var accountNumber int
        accountNumber = 30000001

        for {
                // close the channel after 5 numbers are requested
                if accountNumber > 30000005 {
```

```go
                close(accountNumberChannel)
                return
            }
            accountNumberChannel <- accountNumber
            // increment the account number by 1
            accountNumber += 1
        }
}

func main() {
        accountNumberChannel := make(chan int)
        // start the goroutine that generates account numbers
        go generateAccountNumber(accountNumberChannel)

        // slice containing new customer names
        newCustomers := []string{"SMITH", "SINGH", "JONES", "LOPEZ",
                                 "CLARK", "ALLEN"}

        // get a new account number for each customer
        for _, newCustomer := range newCustomers {
            // is there anything to retrieve from the channel?
            accnum, ok := <-accountNumberChannel
            if !ok {
                fmt.Printf("%s: No number available\n",
                                                  newCustomer)
            } else {
                fmt.Printf("%s: %d\n", newCustomer, accnum)
            }
        }
}
```

SMITH: 30000001

SINGH: 30000002

JONES: 30000003

LOPEZ: 30000004

CLARK: 30000005

ALLEN: No number available

Buffered channels

In the examples in Code Listings 42 and 43, the **accountNumberChannel** can be accessed anywhere in the program to generate the next account number in the sequence. The **generateAccountNumber()** function will not write a new account number to the channel until the existing value has been read by the program. This is because, by default, channels are un-buffered and are therefore synchronous in execution.

Compare the examples in Code Listings 42 and 43 to that in Code Listing 41. In Code Listing 41, there was no attempt to read a value from the channel (using the **myVariable := <- myChannel** syntax). No blocking occurs, and therefore the program continues to generate random numbers and add them to the channel.

If we want, we can buffer the channel, allowing for asynchronous execution. We do this by specifying a capacity for the channel as a second parameter:

myChannel := make(chan *type, capacity***)**

Let's compare the operation of unbuffered and buffered channels. First, the unbuffered channel:

Code Listing 44

```
package main

import (
        "fmt"
)

func main() {
        // un-buffered channel
        c := make(chan int)
        c <- 3
        fmt.Println("OK.")
}
```

fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:

main.main()

      .../src/hello/main.go:10 +0x60

exit status 2

The program terminates with an error because the channel is unbuffered, and we're attempting to put a value into the channel when there is nothing in the program to read from it.

If we create a buffered channel with a capacity of 1, we can add a single value to it without any issues:

Code Listing 45

```
package main

import (
        "fmt"
)
```

```go
func main() {
        // buffered channel, capacity 1
        c := make(chan int, 1)
        c <- 3
        fmt.Println("OK.")
}
```

OK.

Writing to a buffered channel does not block if there is sufficient capacity in the channel. If we attempt to put more values in the channel than we have capacity for, we will get the same error as we saw previously:

Code Listing 46

```go
package main

import (
        "fmt"
)

func main() {
        // buffered channel, capacity 3
        c := make(chan int, 3)
        c <- 3
        c <- 4
        c <- 5
        c <- 6
        fmt.Println("OK.")
}
```

fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:

main.main()

    .../src/hello/main.go:13 +0xf1

exit status 2

Communicating on multiple channels

It is unlikely that a real-life Go program will rely upon a single channel. More typically, there will be multiple channels, and your program will try to coordinate reads and writes across all those channels.

Consider this alternative implementation of our numbers station program. The program listens on two channels: one that generates a sequence of numbers, and the other that waits until it receives a message telling it to stop transmitting. The **broadcast()** function uses a **select** statement to determine if anything is listening on the **nsChannel** (in which case it generates the next number in the sequence) or if it has received a message on the **cChannel** telling it to terminate (in which case it terminates the transmission).

```
select {
case nsChannel <- numbers[i]:
      i += 1
      if i == len(numbers) {
            i = 0
      }
case <-cChannel:
      cChannel <- true
      return
}
```

I have commented liberally throughout this program. See if you can work out what's going on.

Code Listing 47

```go
package main

import (
      "fmt"
      "time"
)

func broadcast(nsChannel chan int, cChannel chan bool) {

      numbers := []int{
            101,
            102,
            103,
            104,
            105,
            106,
            107,
            108,
            109,
            110,
      }
      i := 0
      for {
```

```go
            // see which channel has items
            select {
            /* if the numbersChannel is being listened to,
             take each number sequentially from the
            slice and put it into the channel */
            case nsChannel <- numbers[i]:
                    i += 1
                    /* if we've reached the last number and
                    the channel is still being listened to,
                    start reading from the beginning of the
                    slice again */
                    if i == len(numbers) {
                            i = 0
                    }
            /* if we receive a message on the
            complete channel, we stop transmitting */
            case <-cChannel:
                    cChannel <- true
                    return
            }
        }
}

func main() {
      numbersStation := make(chan int)
      completeChannel := make(chan bool)

      // execute broadcast in a separate thread
      go broadcast(numbersStation, completeChannel)

      // get 100 numbers from the numbersStation channel
      for i := 0; i < 100; i++ {
          // delay for artistic effect only
          time.Sleep(100 * time.Millisecond)
          // retrieve values from the channel
          fmt.Printf("%d ", <-numbersStation)
      }

      /* once we have received 100 numbers,
      send a message on completeChannel
      to tell it to stop transmitting */
      completeChannel <- true

      /* don't terminate the program until
      we receive a message on the completeChannel.
      Discard the response. */
      <-completeChannel

      /* we only get to here if we received a
```

```go
        message on completeChannel */
        fmt.Println("Transmission Complete.")
}
```

101 102 103 104 105 106 107 108 109 110 101 102 103 104 105 106 107 108 109
110 101 102 103 104 105 106 107 108 109 110 101 102 103 104 105 106 107 108
109 110 101 102 103 104 105 106 107 108 109 110 101 102 103 104 105 106 107
108 109 110 101 102 103 104 105 106 107 108 109 110 101 102 103 104 105 106
107 108 109 110 101 102 103 104 105 106 107 108 109 110 101 102 103 104 105
106 107 108 109 110 Transmission Complete.

Bringing it all together

Here is another example of using goroutines and channels, but this one brings together a number of concepts that we have covered in this book. Some of the code here is not immediately obvious, so take the time to work through it, as doing so will help you to fully assimilate some of the features of Go you have learned so far.

Examine the following code and then read the explanation that follows.

Code Listing 48

```go
package main

import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "sort"
)

type WebPage struct {
        URL  string
        Size int
}

type WebPages []WebPage

// implementing the sort.Interface interface in WebPages
func (slice WebPages) Len() int {
        return len(slice)
}

func (slice WebPages) Less(i, j int) bool {
        // Sort of size of response in descending order
        return slice[i].Size > slice[j].Size
```

```go
}

func (slice WebPages) Swap(i, j int) {
	slice[i], slice[j] = slice[j], slice[i]
}

// method for adding a new WebPage element to WebPages
func (wp *WebPages) addElement(page WebPage) {
	*wp = append(*wp, page)
}

// called as a goroutine to retrieve the length of each webpage
func getWebPageLength(url string, resultsChannel chan WebPage) {
	res, err := http.Get(url)
	if err != nil {
		log.Fatal(err)
	}
	defer res.Body.Close()

	// get the size of the response body
	size, err := ioutil.ReadAll(res.Body)
	if err != nil {
		log.Fatal(err)
	}

	// populate the WebPage struct and add it to the channel
	var page WebPage
	page.URL = url
	page.Size = len(size)
	resultsChannel <- page
}

func main() {
	urls := []string{
		"http://www.syncfusion.com",
		"http://www.google.com",
		"http://www.microsoft.com",
		"http://www.apple.com",
		"http://www.golang.org",
	}

	// create a channel
	resultsChannel := make(chan WebPage)

	// call a goroutine to read each webpage simultaneously
	for _, url := range urls {
		/* initiate a new goroutine for each URL
		so that we can analyze them concurrently */
		go getWebPageLength(url, resultsChannel)
```

```
        }

        // store each WebPage result in WebPages
        results := new(WebPages)
        for range urls {
                result := <-resultsChannel
                results.addElement(result)
        }

        // sort using the implementation of sort.Interface in WebPages
        sort.Sort(results)

        // display the results to the user
        for i, page := range *results {
                fmt.Printf("%d. %s: %d bytes.\n", i+1, page.URL,
                                                   page.Size)
        }

}
```

1. **http://www.syncfusion.com: 108794 bytes.**

2. **http://www.microsoft.com: 79331 bytes.**

3. **http://www.apple.com: 31204 bytes.**

4. **http://www.google.com: 19850 bytes.**

5. **http://www.golang.org: 7856 bytes.**

The program examines the length of five webpages and lists them in order of largest to smallest. It executes a goroutine for each webpage so that we don't have to wait to download one webpage before we start on the next. The goroutine pushes the result of each analysis onto a channel, **resultsChannel**.

First, let's see what's new. To get the length of the webpages, we're using Go's **net/http** package. To retrieve the webpage, we call **http.Get()**, passing in the URL of the page we're interested in, and we examine the body of the response. If anything goes wrong, we use Go's **log** package to report a fatal error condition.

Once we're done with the response body, we must close it, and here we're using a **defer** to **res.Body.close()**. The **defer** statement pushes a function call onto a list that Go maintains internally. The function deferred only gets called when the surrounding function completes. You often see **defer** used to simplify functions that do some sort of cleanup.

```
res, err := http.Get(url)

if err != nil {

        log.Fatal(err)

}

defer res.Body.Close()
```

With the body of the response, we can get its length in bytes by reading it into memory with the **ioutil** package's **ReadAll()** method, which returns the number of bytes read:

```
// get the size of the response body
size, err := ioutil.ReadAll(res.Body)
if err != nil {
        log.Fatal(err)
}
```

Everything else in the program builds on what we have learned so far.

In order to store the results for later display to the user, we are creating a couple of custom types. One is called **WebPage** and contains the URL and the length (in bytes) of the response. The other is called **WebPages** and is a slice that contains five instances of **WebPage**, one for each webpage.

Because we want to order the results by size, and the size information is a field buried within our **WebPage** struct, we've implemented the **sort.Interface** interface in the **WebPages** type. To do that, we have included the following methods in **WebPages**:

```
func (slice WebPages) Len() int {
    return len(slice)
}


func (slice WebPages) Less(i, j int) bool {
    return slice[i].Size > slice[j].Size;
}


func (slice WebPages) Swap(i, j int) {
    slice[i], slice[j] = slice[j], slice[i]
}
```

Because we have defined our own type **WebPages**, we have lost the ability to **append()** items to it, which we would keep if we were using a normal slice. So we have created a method on **WebPages** that allows us to add a **WebPage** to the collection when we retrieve it from the **resultsChannel**. Because we are directly changing the contents of **WebPages**, we must use a pointer to the receiver in our method declaration so that we are not simply working on a copy of of it.

```
func (wp *WebPages) addElement(page WebPage) {
    *wp = append(*wp, page)
}
```

Once we have populated **WebPages** with the results of reading the responses from all five websites, we call **sort.Sort()** to retrieve each of the results in size order from largest to smallest and display them to the user:

```
sort.Sort(results)

for i, page := range *results {

    fmt.Printf("%d. %s: %d bytes.\n", i, page.URL, page.Size)

}
```

Chapter 10 Standard Packages

One of the cardinal rules of programming is to reuse code whenever and wherever possible. Why spend time writing code for everyday situations when a bunch of crazily clever people have already done so (and have seen the fruits of their labors tried and tested in the wild)?

So let's have a look at a few of the standard packages that ship with Go. These cover common scenarios that programmers face all the time.

Bear in mind, however, that this is only the tip of the iceberg. I can't hope to cover all the standard packages in Go, or even all features from specific packages. To discover more, use your old friend **godoc** *package_name*.

The `net/http` package

We saw some of the capabilities of the **net/http** package in the previous chapter. However, **net/http** provides a full HTTP client and server implementation. For example, creating a working web server takes just a few lines of code:

Code Listing 49

```go
package main

import (
        "fmt"
        "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello, %s!", r.URL.Path[1:])
}

func main() {
        http.HandleFunc("/", handler)
        http.ListenAndServe(":8080", nil)
}
```

Figure 7: A Working Web Server in a Few Lines of Go

There's much, much more to **net/http**. Look out for my forthcoming book, *Go Web Development Succinctly*, for a deep dive into **net/http**.

You can also see more about the **net/http** package [here](#).

Input and output

A very common requirement in any language is the ability to read from and write to files. Go covers this with the **io** and **io/util** packages on the I/O end of things, and the **os** package to work with files in an operating system-specific manner and provide other operating system functionality.

In fact, you can read and write files with just the functions in the **os** package, but **io** and **io/util** provide a nice wrapper around these for easy use. Augment these with **bufio** functions for buffered reads and writes.

See the following links for more information:

- [io](#)
- [io/util](#)
- [bufio](#)
- [os](#)

Code Listing 50

```go
package main

import (
        "io/ioutil"
        "os"
)

func main() {

        // Write a new file from a byte string
        name := "test.txt"
        txt := []byte("Not much in this file.")
        if err := ioutil.WriteFile(name, txt, 0755); err != nil {
```

```
            panic(err)
        }

        // Read the contents of a file into a []byte
        results, err := ioutil.ReadFile(name)
        if err != nil {
            panic(err)
        }
        println(string(results))

        // Or use os.Open(filename)
        reader, err := os.Open(name)
        if err != nil {
            panic(err)
        }

        results, err = ioutil.ReadAll(reader)
        if err != nil {
            panic(err)
        }
        reader.Close()
        println(string(results))
}
```

> **Note: A common use of** `panic()` **is to abort if a function returns an error value that we cannot (or, in this example, will not) handle.**

Strings

Programmers manipulate strings all the time. We saw some of the things we could do with strings in Chapter 5, but for a heap of new capabilities, look at the **strings** package.

Code Listing 51

```
package main

import (
        "fmt"
        "strings"
)

func main() {
        fmt.Println(
                // does "test" contain "es"?
                strings.Contains("test", "es"),
```

```go
        // does "test" begin with "te"?
        strings.HasPrefix("test", "te"),

        // does "test" end in "st"?
        strings.HasSuffix("test", "st"),

        // how many times is "t" in test?
        strings.Count("test", "t"),

        // at what position is "e" in "test"?
        strings.Index("test", "e"),

        // join "input" and "output" with "/"
        strings.Join([]string{"input", "output"}, "/"),

        // repeat "Golly" 6 times
        strings.Repeat("Golly", 6),

        /* replace "xxxx" with the first two
           non-overlapping instances of "a" replaced by "b" */
        strings.Replace("xxxx", "a", "b", 2),

        /* put "a-b-c-d-e" into a slice using
           "-" as a delimiter */
        strings.Split("a-b-c-d-e", "-"),

        // put "TEST" in lower case
        strings.ToLower("TEST"),

        // put "TEST" in upper case
        strings.ToUpper("test"),
    )
}
```

true true true 2 1 input/output GollyGollyGollyGollyGollyGolly xxxx [a b c d e] test TEST

See more of the **strings** package here.

Errors

We've already seen error messages in Go, but what we don't know yet is that we can create our own errors using the **New()** function in the **errors** package.

The **fmt** package formats an error message by implicitly calling its **Error()** method, which returns a string.

```
package main

import (
        "errors"
        "fmt"
)

func main() {
        err := errors.New("error message")
        fmt.Println(err)
}
```

error message

See more about the errors package [here](#).

Containers

If you come to Go from a language like Java, you might be surprised by the limited number of composite types. In vanilla Go, we have only maps, slices, and channels.

If none of these are suitable for your purposes, check out the **container** package, which introduces three more containers:

- **list**: an implementation of a doubly linked list that uses the empty interface for values
- **ring**: an implementation of circular "list"
- **heap**: an implementation of a "mini heap" in which each node is the "minimum" element in its sub-tree

Code Listing 53

```
package main

import (
        "container/heap"
        "container/list"
        "container/ring"
        "fmt"
)

// following are for container/heap
type OrderedNums []int

func (h OrderedNums) Len() int {
        return len(h)
```

```go
}
func (h OrderedNums) Less(i, j int) bool {
	return h[i] < h[j]
}
func (h OrderedNums) Swap(i, j int) {
	h[i], h[j] = h[j], h[i]
}
func (h *OrderedNums) Push(x interface{}) {
	*h = append(*h, x.(int))
}
func (h *OrderedNums) Pop() interface{} {
	old := *h
	n := len(old) - 1
	x := old[n]
	*h = old[:n]
	return x
}
// end declarations for container/heap

func main() {
	// *** container/list ***
	l := list.New()
	e0 := l.PushBack(42)
	e1 := l.PushFront(11)
	e2 := l.PushBack(19)
	l.InsertBefore(7, e0)
	l.InsertAfter(254, e1)
	l.InsertAfter(4987, e2)

	fmt.Println("*** LIST ***")
	fmt.Println("-- Step 1:")
	for e := l.Front(); e != nil; e = e.Next() {
		fmt.Printf("%d ", e.Value.(int))
	}
	fmt.Printf("\n")

	l.MoveToFront(e2)
	l.MoveToBack(e1)
	l.Remove(e0)

	fmt.Println("-- Step 2:")
	for e := l.Front(); e != nil; e = e.Next() {
		fmt.Printf("%d ", e.Value.(int))
	}
	fmt.Printf("\n")

	// *** container/ring ***
	// create the ring and populate it
	blake := []string{"the", "invisible", "worm"}
```

```go
    r := ring.New(len(blake))
    for i := 0; i < r.Len(); i++ {
        r.Value = blake[i]
        r = r.Next()
    }

    // move (2 % r.Len())=1 elements forward in the ring
    r = r.Move(2)

    fmt.Printf("\n*** RING ***\n")
    // print all the ring values with ring.Do()
    r.Do(func(x interface{}) {
        fmt.Printf("%s\n", x.(string))
    })

    // *** container/heap
    h := &OrderedNums{34, 24, 65, 77, 88, 23, 46, 93}

    heap.Init(h)

    fmt.Printf("\n*** HEAP ***\n")
    fmt.Printf("min: %d\n", (*h)[0])
    fmt.Printf("heap:\n")
    for h.Len() > 0 {
        fmt.Printf("%d ", heap.Pop(h))
    }
    fmt.Printf("\n")
}
```

*** LIST ***

-- Step 1:

11 254 7 42 19 4987

-- Step 2:

19 254 7 4987 11

*** RING ***

worm

the

invisible

*** HEAP ***

min: 23

heap:

23 24 34 46 65 77 88 93

See the following links for more information:

- [list](#)
- [ring](#)
- [heap](#)

Hashes and cryptography

Hashing involves applying a hashing algorithm to a data item, known as the *hashing key*, to create a *hash*. Hashes are used so that searching a database can be done more efficiently, data can be stored more securely, and data transmissions can be checked for tampering. Programmers use hashes a lot.

Go provides support for hashes in its **hash** and **crypto** packages. The difference between the hashes created by the **hash** package and those created by **crypto** is that the latter are much harder to reverse and are therefore more suitable for sensitive data.

The hashing algorithms supported are **adler32**, **crc32**, **crc64**, and **fnv**.

Code Listing 54

```
package main

import (
        "fmt"
        "hash/crc32"
        "io"
        "os"
)

func hash(filename string) (uint32, error) {
        // open the file
        f, err := os.Open(filename)
        if err != nil {
                return 0, err
        }
        // always close files you have opened
        defer f.Close()

        // create the hashing algorithm
        h := crc32.NewIEEE()
        // copy the file into the hasher
        /* - copy() parameters are destination,source. It returns
           the number of bytes written, error */
        _, err = io.Copy(h, f)
```

```go
        // did this work?
        if err != nil {
                // no - return zero and the error details
                return 0, err
        }
        // yes, return the checksum and a nil error
        return h.Sum32(), nil
}

func main() {
        // contents of file1.txt: "Have I been tampered with?"
        h1, err := hash("file1.txt")
        if err != nil {
                return
        }
        // contents of file2.txt: "I have been tampered with!"
        h2, err := hash("file2.txt")
        if err != nil {
                return
        }

        if h1 == h2 {
                fmt.Println(h1, h2, "Checksums match - files are
                                                identical")
        } else {
                fmt.Println(h1, h2, "Checksums don't match - files are
                                                different")
        }
}
```

2407730152 3829883431 Checksums don't match - files are different

See the following links for more information:

- hash
- crypto

Chapter 11 Go Further

That brings us to the end of this e-book. I hope you found the information herein valuable, but more than anything else, I hope that I've given you a real appetite to learn more about Go.

The best way to learn Go is to practice it. Think of some small applications you can build and then build them. Use the documentation to find out about packages you think might help you, then dig into them and discover how they really work.

Above all, if you only *think* you understand something you've written in Go, then you probably don't. Every time you write something, ask yourself if you could explain how it works in simple terms to someone else. Better still, grab the nearest person and do just that. Go, like any other language, rewards the inquisitive.

I found Dave Cheney's blog very useful when I was first starting with Go. Dave is a programmer and author from Sydney, Australia, and has been a Go contributor since 2011 and a maintainer since 2012. His blog is a great source of news, tips, and tutorials, and his "Resources for New Go Programmers" post was a great help to me.

I wish you well in all your Go programming endeavors. Feel free to connect with me at mark@marklewin.com if there's anything I can do to help you.